Diss. ETH No. 20868
TIK-Schriftenreihe Nr. 136

Correlating Flow-based Network Measurements for Service Monitoring and Network Troubleshooting

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences

presented by

DOMINIK SCHATZMANN

Master of Science ETH in Electrical Engineering and
Information Technology
born July 30, 1981
citizen of Zurich, ZH

accepted on the recommendation of
Prof. Dr. Bernhard Plattner, examiner
Dr. Wolfgang Mühlbauer, co-examiner
Dr. Xenofontas Dimitropoulos, co-examiner
Prof. Dr. Rolf Stadler, co-examiner

2012

Abstract

The resilience of network services is continuously challenged by component failures, mis-configured devices, natural disasters and malicious users. Therefore, it is an important but unfortunately difficult task of network operators and service administrators to carefully manage their infrastructure in order to ensure high availability. In this thesis we contribute novel service monitoring and troubleshooting applications based on flow-based network measurements to help operators to address this challenge.

Flow-level measurement data such as IPFIX or NetFlow typically provides statistical summaries about connections crossing a network including the number of exchanged bytes and packets. Flow-level data can be collected by off-the-shelf hardware used in backbone networks. It allows Internet Service Providers (ISPs) to monitor large-scale networks with a limited number of sensors. However, the range of security or network management related questions that can be answered directly by using flow-based data is strongly limited by the fact that only a small amount of information is collected per connection. In this work, we overcome this problem by correlating and analyzing sets of flows across different dimensions such as time, address space, or user groups. This hidden information proves very beneficial for flow-based troubleshooting applications.

Using such an approach, we show how flow-based data can be instrumented to effectively support mail administrators in fighting spam. In more detail, we demonstrate that certain spam filtering decisions performed by mail servers can be accurately tracked at the ISP-level using flow-level data. Then, we argue that such aggregated knowledge from multiple e-mail domains does not only allow ISPs to remotely monitor what their "own" servers are doing, but also to develop and evaluate new scalable methods for fighting spam.

To assist network operators with troubleshooting connectivity problems,

we contribute FACT, a system that implements a flow-based approach for connectivity tracking that helps network operators to continuously track connectivity from their network towards remote autonomous systems, networks, and hosts. In contrast to existing solutions, our approach correlates solely flow-level data, is capable of online data processing, and is highly efficient in alerting only about those events that actually affect the studied network. Most important, FACT precisely reports which address spaces are currently not reachable by which clients – an information required to efficiently react on connectivity issues.

In order to introduce such innovative and productive troubleshooting applications, we improved the entire value chain from low-level data processing all the way up to knowledge extraction.

First, we contribute FlowBox, a modular flow processing library. Its design exploits parallelization and is capable of processing large volumes of data to deliver statistics, reports, or on-line alerts to the user with limited delay. In addition we address the challenge of dissecting large data volumes in real-time into service-related flow sets. Our method identifies Server Sockets (SeSs), i.e., communication endpoints that offer a specific network service to multiple clients (e.g., webmail service, Skype, DNS service). Our approach is application-agnostic, copes with errors of flow data (e.g. imprecise timing) and greatly reduces Internet background radiation.

Second, to infer the underlying network application behind a SeS, we show how correlations across flows, protocols, and time can be leveraged for novel techniques to classify the service provided by a SeS. Although we limit our study to the classification of webmail services such as Outlook Web Access, Horde, or Gmail, our approach is very promising for other classes of network applications, too. Furthermore, we discuss how to achieve service classification with reasonable accuracy and coverage, yet limited effort. Our approach is based on the idea that the service classification of a SeS stays stable across multiple flows and across different users accessing the SeS. This allows us to reduce the overall collection and processing efforts by applying a time and address space sampling scheme during the data mining.

We evaluated our work using flow traces collected over the last 8 years at the backbone of an ISP that covers approximately 2.4 million IP addresses. The large-scale nature of our measurements is underlined by high peak rates of more than 80,000 flows per second, 3 million packets per second, and more than 20 Gbit/s of traffic.

Kurzfassung

Kommunikationsnetze werden kontinuierlich durch Ausfälle von Komponenten, falsch konfigurierten Geräte, Naturkatastrophen oder böswilligen Benutzer herausgefordert. Daher ist es eine wichtige, aber leider schwierige, Aufgabe der Netzbetreiber und Service-Administratoren, ihre Infrastruktur sorgfältig zu verwalten bei gleichzeitiger Gewährleistung einer hohen Verfügbarkeit der Dienste. Um den Betreibern zu helfen, diese Herausforderung besser zu meistern, stellt diese Arbeit neue flow-basierte Anwendungen zur Netzwerk-Fehlerdiagnose vor.

Flow-basierte Messdaten wie IPFIX oder NetFlow liefern eine statistische Zusammenfassung der Verbindungen, welche ein Netzwerkelement durchqueren. Solche Flowdaten können von üblicher Hardware an strategischen Punkten in Netzen aufgezeichnet werden und ermöglichen dadurch eine effiziente Überwachung auch von grossen Netzen. Jedoch können nur wenige Fragestellungen direkt mit Flowdaten beantwortet werden, da pro Verbindung nur wenig Information aufgezeichnet wird. Wir überwinden dieses Problem durch das Korrelieren und Analysieren von Gruppen von Flows über diverse Dimensionen wie Zeit, Adressraum oder Benutzergruppen. Dadurch erschliessen wir bisher ungenützte Information für Anwendungen zur Netzwerk-Fehlerdiagnose.

In dieser Arbeit zeigen wir zunächst, wie Flowdaten korreliert werden können, um Mail-Administratoren bei der Bekämpfung von Spam zu helfen. Genauer zeigen wir, dass die Entscheidung eines einzelnen Mailservers, eine Mail zu filtern, in den Flowdaten sichtbar ist. Dies erlaubt es solche Entscheidungen auf der ISP Stufe zu sammeln und zu vergleichen. Das gesammelte Wissen kann nicht nur zur Leistungsoptimierung diverser Einstellungen der eigenen Server, sondern auch für die Entwicklung und Evaluierung neuer skalierbarer Methoden zur Bekämpfung von Spam verwendet werden.

Weiter zeigen wir, wie Flowdaten korreliert werden können, um Netzwerk-Administratoren bei der Identifikation von Konnektivitäts-Problemen zu helfen. Im Gegensatz zu existierenden Lösungen setzt der vorgestellte Ansatz ausschließlich auf Flowdaten. Überdies ist das Verfahren sehr effizient bei der Entdeckung von genau solchen Konnektivitäts-Problemen, welche die Benutzer auch tatsächlich betreffen.

Damit solche innovativen flow-basierten Anwendungen entwickelt werden können, muss die gesamte Wertschöpfungskette von der low-level Datenverarbeitung bis hin zur Sammlung des Wissens verbessert werden.

Dazu präsentieren wir im ersten Teil der Arbeit unseren flow-basierten Ansatz zur Identifizierung von Kommunikationsendpunkten, welche Netzwerkdienste wie z.B. Webmail, Skype oder DNS zur Verfügung stellen. Die Identifikation der Endpunkte ist notwendig, um grosse Mengen an Flowdaten effizient in kleinere,dienstspezifische Mengen zu unterteilen. Die Auswertung des Ansatzes mit Hilfe gesammelter Flow Daten eines grossen Netzes belegt, dass unser Ansatz für die Echtzeit-Verarbeitung von Messdaten geeignet ist.

Im zweiten Teil werden die Eigenschaften von Kommunikationsendpunkten untersucht und Verfahren entwickelt, die den Dienst bestimmen, der von einem Endpunkt angeboten wird. Mithilfe dieser Information kann später die eigentliche Problemdiagnose vereinfacht werden, da man gezielt Flows bestimmter Netzwerkdienste untersuchen kann. Zum Beispiel wird gezeigt, wie die Korrelationen zwischen Benutzern, Protokollen, und Zeitinformationen der Flows genutzt werden kann, um neue Methoden zur Bestimmung des Dienstes zu entwickeln. Obwohl diese Studie auf die Identifizierung von Webmail-Diensten wie Outlook Web Access, Horde oder Google Mail beschränkt ist, ist der Ansatz vielversprechend, auch für andere Dienste. Ferner wird diskutiert, wie sich der Aufwand zur Bestimmung des Dienstes reduzieren lässt, ohne zuviele Details Preis zu geben. Unser Ansatz beruht darauf, dass der gleiche Dienst über eine längere Zeit für verschiedene Benutzer angeboten wird. Somit kann die Bestimmung dieses Dienstes auch anhand einer Stichprobe erfolgen. Durch Zwischenspeichern des Resultats kann eine Wiederholung des ressourcenintensiven Arbeitsschritts der Bestimmung des Diensts verhindert werden.

Wir haben unsere Arbeit mit Hilfe von Flowdaten ausgewertet, welche über die letzten 8 Jahre im Netz eines Schweizer ISPs aufgezeichnet wurden. Dessen Netz beherbergt circa 2.4 Millionen IP Adressen und erreicht Übertragungraten von von mehr 20 Gbit/s und Flowraten von mehr als $80,000$ Flows/s .

Contents

List of Figures

List of Tables

Glossary

Application Classification is the general process of identifying the application causing certain network traffic. An overview of different application identification methods is provided in Chapter 6.

Application Label is the outcome of the application classification process. For example, a DPI appliance labels network flows with the help of payload inspection with application labels.

ARPANET stands for Advanced Research Projects Agency Network and was the world's first operational packet switching network.

FlowBox is our modular toolbox for on-line flow processing and is introduced in Chapter3.

Server Socket is a Socket where a process waits on incoming requests from multiple clients to provide a specific network service such as HTTP, Skype or Bittorent. The concept of Server Sockets is introduced in more detail in Chapter 1.

Service Label is the outcome of the service classification process.

Socket is uniquely identified by IP address, protocol number (6 for TCP, 17 for UDP), and TCP/UDP port number. Each bidirectional flow summarizes data exchange between two sockets.

spam "is the use of electronic messaging systems to send unsolicited bulk messages indiscriminately." [155]

SWITCH is ISP that connects multiple research labs, universities, and government institutions of Switzerland to the Internet.

Acronyms

AMS-IX Amsterdam Internet Exchange

BGP Border Gateway Protocol

CeS Client Socket

CPU Central processing unit

DDoS Distributed denial-of-service attack

DoS Denial-of-service attack

DPI Deep Packet Inspection

ETH Eidgenossische Technische Hochschule

ETHZ ETH Zurich

FTP File Transfer Protocol

GNU GNU Project

HTTP HyperText Transfer Protocol

HTTPS HyperText Transfer Protocol Secure

IETF Internet Engineering Task Force

IP Internet Protocol

ISP Internet Service Provider

MTA Mail Transfer Agent

NANOG North American Network Operators Group

NTP Network Time Protocol

OWA Outlook Web Access

P2P Peer to Peer

POSIX Portable Operating System Interface

RAID Redundant Array of Independent Disks

RAM Random-access memory

RTP Real-time Transport Protocol

SeS Server Socket

SMTP Simple Mail Transfer Protocol

SPF Sender Policy Framework

SSD Solid-state drive

SSH Secure Shell

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLS Transport Layer Security

UTC Coordinated Universal Time

VPN Virtual private network

Chapter 1

Introduction

1.1 Importance of Network Management

At 22:30 hours on October 29, 1969, the two first nodes of the later ARPANET exchanged a data packet representing the single character "l" [1] [150]. 40 years later, this experiment has evolved into the Internet, which is used by billions of people to transport sophisticated information such as text, media, or application data. [62, 144, 151]. Nowadays, the Internet is ubiquitous and is used for personal, business, and government communication. Evidently, it represents a critical infrastructure, similar to water supply, transportation, electricity, gas or oil supply chain [48, 136]. It is not surprising that attacks against the Internet are already seen as an act of war, possibly provoking an armed response [55, 78, 156].

Contrary to public perception, the resilience and availability of computer networks and services is not only put to test by malicious users. Mis-configurations [16], component failures [89, 125], or fiber cuts [83, 138] can lead to network failures and service disruptions. It is an important but unfortunately difficult task for network operators and service administrators to ensure high availability of computer networks and network services. One major motivation behind this thesis is to improve the state-of-the art in network management leading to better resilience.

More than 40 years of network research have passed, and the reader may

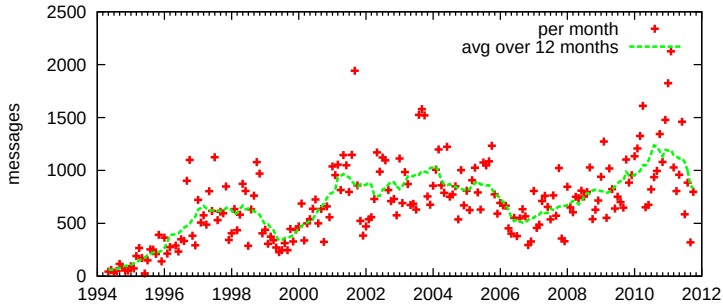
[1] This was the first character of the word login. Unfortunately, the experimental network crashed during the transmission of the letter "o".

ask if not all research problems in the area of network management or network resilience have been solved and addressed? Our clear answer is: no!

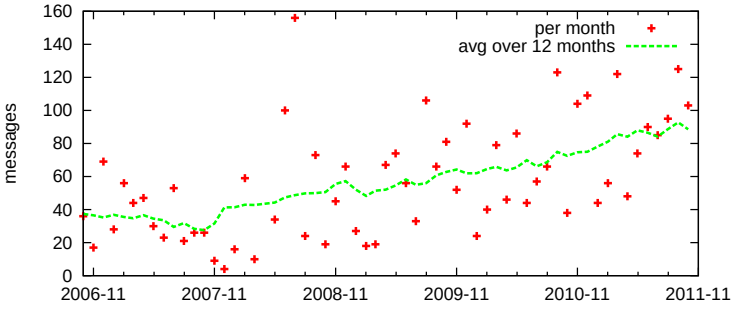
For example, right from the beginning the electronic mail service has been constantly abused by malicious users to distribute unsolicited bulk messages known as spam [155] over the network. In fact, according to IronPort's 2008 Security Trend Report [63], as much as 90% of inbound mail is classified as spam. Without continuous improvement of countermeasures to protect our e-mail inboxes, the electronic mail service would quickly become totally useless. Interestingly, Rao et al. have estimated in [114] that the cost society has to pay for fighting spam is even higher than the worldwide financial damage due to vehicle theft. To assist email service administrators in this ongoing fight against spammers, we present in this thesis a method to track spam based on network measurement data collected in the Internet backbone. By passively collecting this knowledge in the network core, we establish the ground for new applications that can exploit the scope of multiple independent mail servers to fight spam. For instance, this knowledge can be used to accumulate network-wide spam statistics, to compare the spam filtering performance of the different email servers, or to build a passive collaborative spam filter. As we show in Chapter 8, this method can be used to improve the resilience of the electronic mail service.

Furthermore, providing end-to-end reachability in the Internet apparently still is a major issue for network administrators. For example, the network operators of the SWITCH network observed on March 25th, 2010, the following connectivity problem: After a scheduled maintenance by the Amsterdam Internet Exchange (AMS-IX), the connection of SWITCH to this exchange point came only back with partial connectivity, with the traffic towards a few external network prefixes being dropped. Several customers complained about external services being unreachable. Overall, it took more than four hours until the problem was finally resolved by resetting a port of the router. Such incidents manifest the need for better tools that allow to monitor and troubleshoot connectivity and performance problems in the Internet. To assist network operators to improve the resilience of their networks we contribute in this thesis a method to monitor connectivity with remote autonomous systems, subnets, and hosts based on network measurement data collected in the Internet backbone. As we discuss in Chapter 9, our approach would have helped the operator to detect such a serious problem much faster and would have provided valuable hints about the origin of the problem.

Interestingly, this kind of end-to-end reachability incidents occur more



(a) NANOG



(b) OUTAGE

Figure 1.1: *Number of messages on NANOG and OUTAGE mailing list.*

often than commonly believed. Even today, operator forums are full of reports about temporary unreachability of complete networks. To estimate the impact of our method, we have analyzed the number of e-mails posted per month in the North American Network Operators Group (NANOG) mailing list over the last 18 years. The findings of this analysis, shown in Figure 1.1(a) suggest that there has been no decrease over the last years. After a linear rise between 1994 and 1998, the number of messages per month stays stable at high level between 500 and 1,000 messages per month. We repeated this experiment studying the Outage Mailing List [105] between 2006 and 2011. As illustrated in Figure 1.1(b), the average number of mails posted per months has even increased from 40 to 80 mails over the last 5 years. This underlines the importance of this research.

1.2 What Network Data Should be Measured?

The statement "You can't manage what you can't measure" gets it to the point. Network operators need to perform network measurements. Only then, they can timely react when patterns of network usage change [90, 91], or plan the further development of their networks.

As first step, it is crucial to decide about the granularity of data to be collected and to be processed to build network management applications such as discussed in the last section. As shown in Figure 1.2, we can distinguish between three basic types of data sources that differ in their granularity, in their coverage, and in their expressiveness.

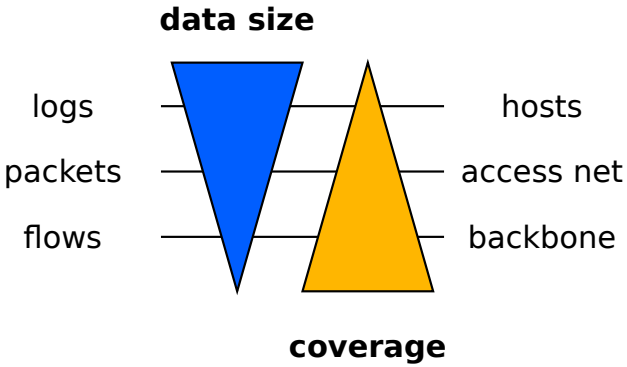


Figure 1.2: *Information trade-off of measurement data.*

Log files: coverage of individual hosts

Sensors on individual hosts at the network edge can record detailed information about the activity of a certain network service. Compared to the other data types, such sensors allow to include meta information that is not directly included within the network traffic. For instance, a sensor on a host providing a mail service is able to log if the recipient address of a incoming mail exists at all or if the received mail was classified as spam.

Today, host-based sensors are deployed on servers [101], workstations [77],

and even mobile phones [21]. Their scope is restricted to a single entity and application. Achieving high network-wide coverage would require to deploy a huge number of sensors if possible at different locations inside the network. Tremendous efforts and privacy concerns [17] practically rule out such an approach. For example, to make network-wide statements on the overall byte volume caused by spam e-mails, it would be necessary to collect, analyze, and combine log files from *all* mail servers inside the respective network. However, since e-mail logs include highly sensitive information, e.g., who talked to whom and about what, email administrators are generally not willing and not allowed to share this information at such a fine granularity.

Packet-level: coverage of small-sized access networks

At gateway devices of edge or access networks, sensors can be deployed that analyze the inbound and outbound traffic of a set of hosts. Typically, such sensors mirror the observed packets. The obtained traces include packet headers and payload.

The major drawback are privacy concerns and scalability issues: large-scale collections of packet-level traces aligned with automated, comprehensive analysis of packet payloads raise privacy concerns. In addition, the volume of collected data is often large, which requires expensive customized processing and storage solutions. A major challenge is to save the packet data at line speed to disk and to keep the data of larger observation intervals for later offline analysis. However, on infrastructures where traffic of interest is not encrypted (by policy or because keys are known to the third party) and where data volume is limited, packet-level data provides a rich source of information. It can be inspected e.g., by researchers to develop new protocols, or network administrators to delay or block certain types of network traffic in real-time, or to debug network performance issues such as TCP throughput problems.

Flow-level: coverage of large-scale networks

To monitor the core of networks (“backbone”), network operators deploy sensors that export only aggregate statistics about observed network connections, i.e., *flows* [7, 30]. Typically, statistics about the total number of bytes and packets that are exchanged between two sockets are recorded. This approach requires only a few bytes of storage per connection. For example, to download the source code of the Linux kernel, several hundred megabytes of data

need to be transferred over the network, resulting in thousands of packets. Yet, the respective flow record will only store the IP addresses of the user and the webserver, as well as the number of exchanged bytes and packet. This means that the size of measured data is independent of the actually transfered number of bytes.

This light-weight property of flow data makes network measurement scale very well and allows to monitor even very large network segments with only a few sensors.

Flow-based approach – the rising star

The Internet experiences a continuous and rapid growth. More and more Internet services are offered that rely on distributed service architectures, nowadays widely referred to as the "cloud". Changes in the service landscape are likely to have an impact on the global traffic patterns of future networks. In our eyes, it will be indispensable to revisit flow-level approaches to network measurement and monitoring for the following reasons:

First, more and more application logic will be outsourced to remote servers or into the cloud. For example, already today, Internet users rely on word processing or spreadsheet applications such as Google Docs [53]. Therefore, the number of local servers that could potentially provide log data to the local network operator is likely to shrink in the near future.

Second, the traffic volume injected by end hosts will rise. Since application data and application logic itself is stored in the cloud rather than on individual hosts, the end hosts frequently pull data from remote servers and push data back after modification. For packet-level sensors, given their poor scalability, it is questionable whether they can cope with the growth of Internet traffic. Furthermore, despite recent findings that packet-level capturing up to 10 Gbits is feasible [13], the processing of the voluminous data to apply state-of-the-art traffic classification algorithms is challenging due to today's IO bandwidth limitation, memory access times, and storage space needed.

Third, the fraction of encrypted traffic is expected to increase. Users become more privacy-aware [8, 41, 79, 131] and protect their data using HyperText Transfer Protocol Secure (HTTPS) instead of HyperText Transfer Protocol (HTTP). The first herald of these changes is already visible: cloud-based storage services like Dropbox, Ubuntu Cloud, or Apple's iCloud rely on HTTPS [3, 38, 65]. Furthermore, default access for Gmail and Google Search was changed to HTTPS in January 2010 [129] and March 2012, re-

spectively. Many popular websites such as Facebook [115] or Twitter [148] provide an optional HTTPS interface. To ensure privacy, browser plugins are installed that automatically redirect the browser to such optional HTTPS interfaces [40]. Therefore, the captured information from packet-level sensors will be of a little value if the fraction of encrypted traffic further increases.

Fourth, only flow-level data can cover large-scale networks (though at a coarser granularity) and allow us to study the network as a whole. Using packet-level sensors to monitor networks of similar size would cause tremendous costs. However, without having the full overview of the network, troubleshooting applications such as the discussed end-to-end reachability tracking are hard to realize.

Therefore, we believe that flow-based approaches are the right choice and dedicate our thesis to flow-based network measurement and monitoring.

1.3 Grouping Flows along Communication Endpoints

As discussed, flow-based measurements are a promising source of information for large-scale network monitoring and service troubleshooting. However, the explicit information encoded in flows is restricted to counters such as the number of exchanged packets or the number of exchanged bytes for each observed network connection.

Due to these limitations people often assume that flow-based methods are restricted to plain traffic accounting or traffic engineering [27, 84]. However, recent research results in the field of flow-based network anomaly detection [140, 141] demonstrated that flow-based methods using generalized entropy measurements can be used to accurately detect and even determine the type of an anomaly (such as DoS, DDoS, scan attacks). This progress has been achieved by calculating the generalized entropy over larger flow sets. Importantly, such results illustrate that important information can be hidden within flow sets, a finding, which we will leverage within this thesis.

Our ultimate goal is to widen the class of applications in the area of network troubleshooting that can benefit from flow-based network measurements. To achieve this goal, we plan to exploit **information in sets of flows**. However, instead of partitioning flows by time interval, we propose to group flows together if they belong to the **same communication endpoint** that offers a specific network service to multiple clients. Such service-oriented flow

sets can be analyzed across dimensions such as time, address space, or users to access unused information that is often overlooked.

Evidently, determining and analyzing flow sets first requires to define the notion of communication endpoints in more detail. Over the years many different definitions were introduced to describe communication between endpoints such as network applications, network services, servers and clients or peer-to-peer services. Being aware of this, we propose to use a notation that is more inspired by the actual implementation of the low-level network communication than the type of service provided by the endpoint. More precisely, we propose to use the notation of a Socket as an endpoint for network communication between two machines as used by network programming libraries such as Berkeley sockets, POSIX sockets or Java.Net.Sockets:

Definition 1 (Socket) *A socket is uniquely identified by IP address, protocol number (6 for TCP, 17 for UDP), and TCP/UDP port number. Each bidirectional flow summarizes data exchange between two sockets.*

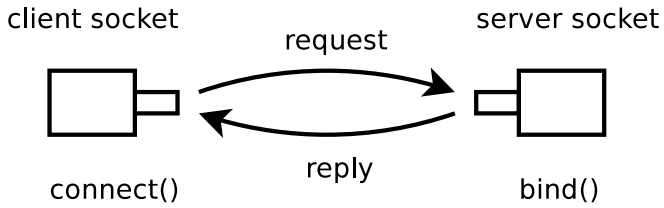


Figure 1.3: *The concept of Server Sockets.*

Furthermore, we can classify a Sockets involved in a communication according to its behavior into Client Sockets (CeSs) and SeSs as illustrated in Figure 1.3. A SeS is a Socket where a process waits for incoming requests from other Sockets to provide a specific service (e.g., webmail service, Skype, DNS service). Typically, the process on this host implements the `bind()` function call to wait for incoming requests. A CeS is a Socket that is opened by a process to communicate with a specific SeSs.

Definition 2 (Server Socket (SeS)) *A SeS is a Sockets where a process waits on incoming requests from clients to provide a specific service.*

By grouping all flows belonging to the same SeS into one set we assure

that flows belonging to the same "application" are grouped together. This is the foundation to successfully correlate and analyze groups of flows.

Using these definitions to dissect flow-level measurement data into flow sets, we make minimal assumptions about the specific type of a network service . This is important since during the data processing the actuale "application" causing the flow is most likely unknown to the system.

We like to pinpoint that in principle, our definitions can capture both classical services (web, e-mail, FTP, SSH, etc.) and P2P-like communication. The latter is captured by our definition, since each P2P client provides a service socket, used by other P2P clients to contact this specific node. Therefore this definition captures P2P clients as well as P2P supernodes such as Skype clients or Skype supernodes. Our only implicit assumption is that each SeS only offers one network service at the same time. This is reasonable since two processes generally cannot listen on the same protocol port pair. Typically the system would answer such a request with an "Address already in use" error message.

Overall, we will show in this thesis that grouping flows based on SeSs allows to efficiently summarize monitored traffic of large networks and build new types of troubleshooting applications.

1.4 Bird's Eye View on this thesis

The art of implementing network monitoring applications is to efficiently extract from very large data sets the required information that is beneficial to the network operator. Flow sets can be very useful in this context.

This challenge can be seen as one explicit instance of the more general "DIKW Hierarchy" model also known as "Knowledge Pyramid" used in information management or information systems to describe structural and/or functional relationships between data, information and knowledge.

In this model, Data, Information, and Knowledge are aggregated within in a pyramid, as illustrated in Figure1.4. This arrangement is based on the strong belief that knowledge can only be built on information and information is a product of processed data [119]. In this general model, "data" can mean objective observations or "chunks of facts about the state of the world" [47]. In our context this corresponds to flow-based measurement data collected by the sensors. Within this model "information" is "inferred from data" [120] and is refereed as the process of making the data "useful" for other applications.

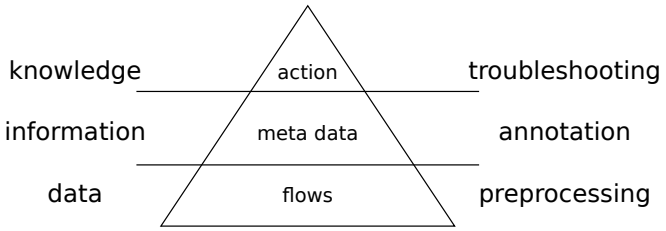


Figure 1.4: *The knowledge pyramid.*

In the context of our flow-based measurements, this transformation corresponds to the processing step where flow sets sets are annotated with meta data required for the decision making process. It is agreed that "knowledge" is "difficult to define" in general [120]. However, it can be seen as "information having been processed, organized or structured in some way, or else as being applied or put into action" [152]. In our context we can think of a troubleshooting application that arranges the information to inform the operator about the state of the network or a service.

This let us to conclude that to provide operators innovative and productive network and service troubleshooting applications, it is not enough to improve only a single part of the Knowledge Pyramid. Therefore, in this thesis, we intentionally focused on all three phases.

The rest of this subsection provides a short outlook on each phase and introduces the related research questions and highlights our main contributions. In addition, we briefly summarize the related work of the corresponding research area. More details about the individual topics are provided in the respective chapters, see Figure 1.5. Further, a compact summarize of the contributions can be found in Chapter 10.

1.4.1 Preprocessing

To achieve reasonable results, it is necessary to first identify and eliminate inconsistencies within the measurement data that are caused by the measurement infrastructure. Furthermore, as we need to cope with voluminous measurement data streams, typically caused by large networks, the incoming data stream should be restricted to the essential parts in order to focus on the analysis of a specific question. Finally, aggregating flows into flow set data structures is required to simplify the further data processing.

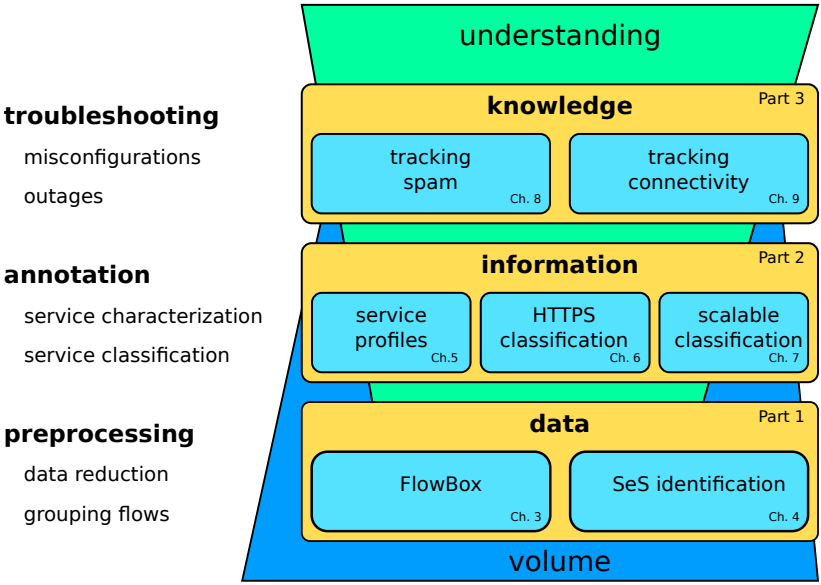


Figure 1.5: *Bird's eye view of this thesis.*

In the context of data preprocessing, we present in Chapter 3 our toolbox for on-line flow processing called FlowBox. It is designed to exploit parallelization to address the challenge of processing voluminous data within limited time to timely deliver statistics and reports to the user. To reduce the time required to build a new analysis or alerting system, FlowBox is organized in loosely coupled modules that can easily be reused. Additionally, FlowBox provides an interface written in Ruby, which allows users to quickly evaluate new ideas in a comfortable way.

Based on FlowBox we present in Chapter 4 our preprocessing chain to identify the SeSs. This chain includes for example a set of validation checks that systematically exclude corrupted data from further processing. Further to reduce the data volume without losing relevant information, we describe in Section 4.3.2 how to eliminate duplicated flows that are due to the fact that some flows are visible at two vantage points of our measurement infrastructure. In our study, this filtering step reduces the amount of flow records by one third, which significantly increases the performance of the general data processing. In Chapter 4 we present our data-driven method to identify the SeSs.

This approach continuously analyzes the flow-level communication patterns to identify popular communication endpoints. This allows to identify new services and track them while limiting the amount of information about detected services that needs to be kept in memory. Our proposal copes with impreciseness of flow data (e.g. imprecise timing) , eliminates Internet background radiation and is able to cope with peak rates of more than 20 Gbit/s.

With respect to service detection, several proposals have been made. Many of them rely on active probing, e.g., [9,85] instead of passively collected measurement data as in this thesis. In contrast to a packet-level solution [149], our approach is based on flow-level data and is applicable even for large networks.

In summary, we address the following research question in these chapters: How can we identify communication endpoints in a continuous stream of flow-level network measurement data in a scalable way?

FlowBox FlowBox is released to the community under GNU Lesser General Public License.

SeS Identification Our method to identify communication Endpoints is released within FlowBox, documented in [126] and is already re-used in other research projects [1,5].

1.4.2 Annotation

After preprocessing the data we focus on the question how SeSs can be labeled with service-specific tags (service label). Such a tag can allow the troubleshooting layer to treat the flows belonging to diverse services differently. For example, a troubleshooting application focusing on reachability problems can use this service label to treat incidents from different service types differently. This can be very useful if certain service classes like Skype or P2P File-sharing application have different reachability characteristics than for example webmail services that should be constantly reachable.

Initially, to improve our basic understanding of the "service landscape" of the Internet, we study in Chapter 5, where in the Internet address space SeSs are located, what traffic characteristics they reveal, and how characteristics change over time. Based on these insights, we then address two important topics relating to annotation: First, we present in Chapter 6 a new set of flow features that allow us to classify HTTPS services into fine-granular service groups. This extension is critical since more and more services will rely on encrypted HTTP traffic in the future. Second, we show in Chapter 7 how

to achieve service classification with reasonable accuracy and coverage, yet limited effort. More precisely, we analyze if service labels from a subpart of the network can be projected to the overall network. Our idea is based on the observation that a service label for a given server socket is stable across multiple flows and different users that access the same socket. We exploit this fact to develop a new labeling approach that reduces the overall collection and processing efforts.

Related work in the area of service annotation is clearly related with the research field of traffic characterization and traffic classification. Recently, Internet-wide characterizations of network traffic and application usage have caught public attention [15, 81]. The study by Labovitz et al. [81] has been a milestone towards a better understanding of inter-domain traffic and how it changes over times. Our work nicely complements their work. While Labovitz et al. [81] look at network applications on a coarse level (e.g., aggregates such as P2P or web traffic), we study them at the level of individual SeSs and present the evolution for even the last 8 years. With respect to traffic classification, different approaches were proposed [71]. This includes port-based approaches [99], signature-based solutions [25, 58, 67, 99, 132] based on packet payload analysis, statistics-based [6, 10, 33, 87, 95, 98, 118], or host-behavior-based [61, 67, 68] techniques. However, port-based approaches are insufficient since all HTTPS application use port 443, signature-based approaches fail when payload is encrypted. Further, we found that current statistics-based approach based on per flow-based features are not sufficient to classify individual HTTPS applications. Host-based approaches attempt to classify a host directly based on its communication pattern and are the closest in spirit to our approach of exploring information included in flow sets based on SeS. Yet, host-based approaches have not been extended to HTTP(S) traffic classification.

In summary, we address the following research questions in this part: How can we classify HTTPS services into fine-granular service groups using only flow-level traffic features? And how can we classify services in general with reasonable accuracy and coverage, yet limited effort?

HTTPS classification Our approach to uncover HTTPS webmail applications solely based on flow-level features is presented in [127].

Scalable classification Our idea to develop new labeling methods that reduce the overall collection and processing effort is presented in [128] (under submission).

1.4.3 Troubleshooting

Regarding troubleshooting, the thesis at hand demonstrates two example applications that make use of our flow-based measurement and monitoring approach.

First, we focus on troubleshooting mail services. In Chapter 8, we show that even the limited information encoded in flow-level measurement data is sufficient to reliably identify spam in the Internet backbone. We discuss how this information can be used to identify performance problems of mail servers.

Related work in the area of spam detection and troubleshooting mail services is manifold. Analyzing the content of individual mails to identify spam can cause significant processing, storage, and scalability challenges creating a need to at least perform some fast "pre-filtering" on the email server level. To do this, email servers evaluate information received at various steps of the SMTP session using local (e.g., user database, greylisting [60]) and global knowledge (e.g., blacklists [133, 135] or SPF [158]) to identify and reject malicious messages, without the need to look at the content. Recently, new filtering approaches focusing on general network-level characteristics of spammers are developed [11, 31, 56, 112], which are more difficult for a spammer to manipulate. An example of such characteristics are geodesic distance between sender and recipient [137], round trip time [11] or MTA link graph properties [36, 50]. Our work is in the same spirit, however we look at the problem from a somewhat different perspective. Specifically, we look at the problem from an AS or ISP point of view, comprising a network with a large number of email servers. In this setup we present a 100% passive, minimally intrusive, and scalable network-level method to infer and monitor the prefiltering performance and/or policy of individual servers, and discuss how this local server knowledge can be collected and combined in order to re-use it to improve individual server performance.

Second, in Chapter 9 we try to pinpoint reachability problems towards remote networks. To this end, we propose FACT, a system that implements a **F**low-based **A**pproach for **C**onnectivity **T**racking. This system enables network operators to monitor whether remote hosts and networks are reachable from inside their networks or their customer networks, and to alert about existing connectivity problems. Such issues include cases where either we observe a significant number of unsuccessful connection attempts from inside the studied network(s) to a specific popular remote host, or where many remote hosts within external networks are unresponsive to connection attempts

originated by potentially different internal hosts.

With respect to service and network outage detection, both researchers [69, 88, 161, 162] and industrial vendors [4, 19] have made proposals for detecting and troubleshooting events such as loss of reachability or performance degradation for traffic that they exchange with other external networks, unfortunately with mixed success. Predominantly, such tools rely on active measurements using ping, traceroute, etc. [88, 162]. Besides, researchers have suggested to leverage control plane information such as publicly available Border Gateway Protocol (BGP) feeds [66, 69, 106], although Bush et al. [18] point out the dangers of relying on control-plane information. Compared to existing work, our approach relies on flow-level measurement data (and not on control-plane data), focuses on events that actually affect the monitored network or its users, and and provides valuable hints about the origin of the problem including e.g. a list of the unreachable network address space.

One major concern of the thesis at hand is to translate the obtained insights into real applications that are used by network operators. To this end, we have already deployed a first prototype of FACT. The results are promising and have already fostered new research to extend flow-based connectivity tracking approaches [5, 97].

In summary, we address the following research question in this part: How can spam and connectivity problems toward remote networks be identified using flow-level measurement data?

Tracking spam Our method to track spam using flow-level data from the network core is presented in [124].

Tracking connectivity Our approach to track connectivity towards remote autonomous systems, networks, and hosts is presented in [125].

1.5 Publications

The work presented in this thesis is based on the following publications:

P01: **Digging into HTTPS: Flow-Based Classification of Webmail Traffic**
D. Schatzmann, W. Mühlbauer, T. Spyropoulos, X. Dimitropoulos
Internet Measurement Conference (IMC), 2011

P02: **Inferring Spammers in the Network Core**
D. Schatzmann, M. Burkhart, T. Spyropoulos
Passive and Active Measurement Conference (PAM), 2009

P03: **FACT: Flow-based Approach for Connectivity Tracking**
D. Schatzmann, S. Leinen, J. Kögel, W. Mühlbauer
Passive and Active Measurement Conference (PAM), 2011

P04: **Flow-level Characteristics of Spam and Ham**
D. Schatzmann, M. Burkhart, T. Spyropoulos
TIK Technical Report 291

P05: **Flow-Based Dissection of Network Services**
D. Schatzmann, W. Mühlbauer, B. Tellenbach, S. Leinen, K. Salamatian
TIK Technical Report 338

P06: **Scaling Traffic Classification through Spatio-temporal Sampling**
D. Schatzmann, W. Mühlbauer, T. Spyropoulos, K. Salamatian
TIK Technical Report 349

In addition, these publications were coauthored during this thesis:

P07: **Modelling the Security Ecosystem - The Dynamics of (In)Security**
S. Frei, D. Schatzmann, B. Plattner, B. Trammell
Workshop on the Economics of Information Security (WEIS), 2009.

P08: **The Role of Network Trace Anonymization under Attack**
M. Burkhart, D. Schatzmann, B. Trammell, E. Boschi, B. Plattner
ACM SIGCOMM Computer Communication Review (CCR), 2010.

P09: **Peeling Away Timing Error in NetFlow Data**
B. Trammell, B. Tellenbach, D. Schatzmann, M. Burkhart
Passive and Active Measurement Conference (PAM), 2011.

P10: **Identifying Skype traffic in a large-scale flow data repository**
B. Trammell, E. Boschi, G. Procissi, C. Callegari, P. Dorfinger, D. Schatzmann
Traffic Measurement and Analysis Workshop (TMA), 2011.

P11: **A tale of two outages: a study of the Skype network in distress**
B. Trammell, D. Schatzmann
Int. Workshop on Traffic Analysis and Classification (TRAC), 2011.

P12: **WiFi-Opp: Ad-Hoc-less Opportunistic Networking**
S. Trifunovic, B. Distl, D. Schatzmann, F. Legendre
ACM MobiCom Workshop on Challenged Networks (Chants), 2011.

P13: **Twitter in Disaster Mode: Security Architecture**
T. Hossmann, P. Carta, D. Schatzmann, F. Legendre, P. Gunningberg, C. Rohner
ACM CoNext Special Workshop on the Internet and Disasters, 2011.

P14: **Accurate network anomaly classification with generalized entropy metrics**
B. Tellenbach, M. Burkhart, D. Schatzmann, D. Gugelmann, D. Sornette
Journal of Computer and Telecommunications Networking, 2011.

P15: **On Flow Concurrency in the Internet and its Implications for Capacity Sharing**
B. Trammell, D. Schatzmann
ACM CoNext Capacity Sharing Workshop (CSWS), 2012.

P16: **Horizon Extender: Long-term Preservation of Data Leakage Evidence in Web Traffic**
D. Gugelmann, D. Schatzmann and V. Lenders
ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2013

Part I

Foundations of Large Scale Internet Measurements

Chapter 2

Measurement Setup

Throughout the remainder of this thesis, we rely on flow data collected at the backbone of the SWITCH network, an ISP that connects multiple research labs, universities, and government institutions of Switzerland to the Internet. To enrich this data source and to validate our findings, we additionally use information provided by host-level or access network-level probes such as email server logs or data collected by a Deep Packet Inspection (DPI) appliance.

In this chapter we first introduce the monitoring setup used to collect the flow traces and provide a short high-level characterization of the observed traffic patterns. Then, we describe the installation used to connect the DPI appliance to an access network within the SWITCH network. In addition, we report our findings about a set of manually labelled SeSs. This manual labelling was required since the DPI appliance can only provide limited insights for encrypted HTTP or encrypted mail traffic. Finally, we introduce our log-level data that was collected on a mail server.

2.1 Flow Data

According to student registration and employee statistics, the estimated number of actual users of the SWITCH network amounts to approximately $250,000$ in 2010, and $200,000$ in 2003. The IP address space that is announced via BGP to the Internet currently covers approximately 2.4 million IP addresses and the topology of the backbone is illustrated in Figure 2.1. Since the ad-

dress space has remained relatively stable over the years, we conclude that the network itself has not undergone any significant changes in its user base. Hence, it is ideally suited for studying network services and general changes in their usage even over longer time periods.

Of course every measurement study is biased by the usage of the examined network. The SWITCH network is an academic network and its traffic mix is likely to be different from company or residential networks. However, we like to point out that the presented methods should be applicable to other networks types without any major constraints.

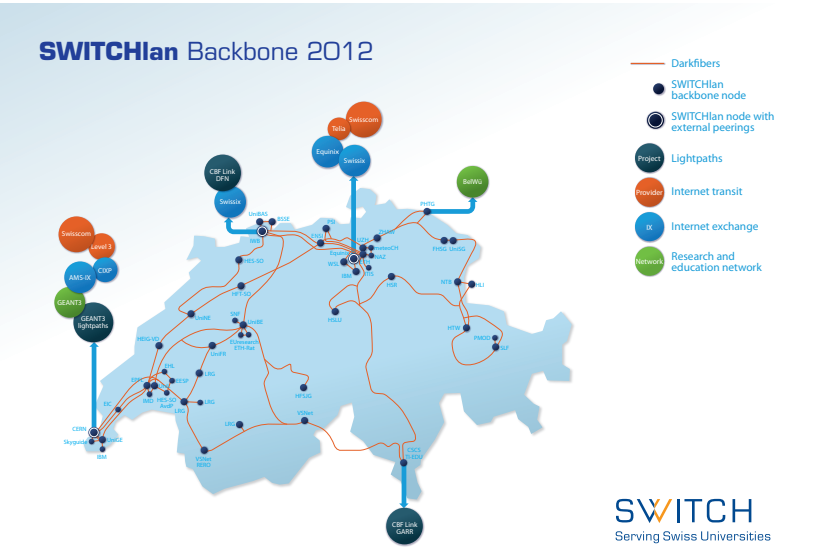


Figure 2.1: *A map of the SWITCH backbone in 2012 (Source: SWITCH).*

Traffic information has been collected in the form of flow-level data from 2003 until today. We record all flows that cross any border router, thus obtaining all traffic that the studied network exchanges with the Internet. To this end, we capture unsampled NetFlow data at all six border routers[1] with a total of 15 upstream and peering links in 2010[2]. To cope with high traffic rates,

[1]Before 2009 the network only had 5 border routers (2008) and 4 border routers (2003-2007), respectively.

[2]2003: 7 external links, 2004: 11 external links, 2005: 12 external links, 2006-2007: 13

we have to rely on hardware-based flow collection [29], using NetFlow in its version 5 until 2008, and in its version 9 afterwards. Due to limitations of the used hardware, information about TCP flags is not available. Finally, we store the obtained flow records at a central data repository. Figure 2.2 visualizes our flow-level data collection infrastructure.

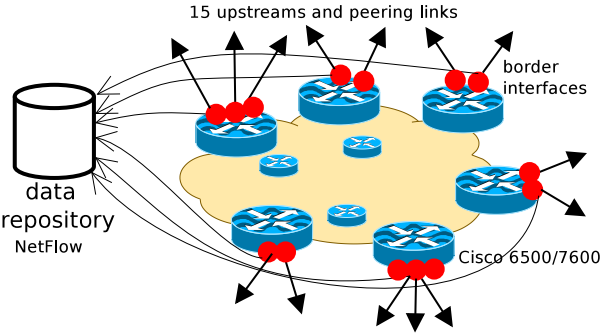
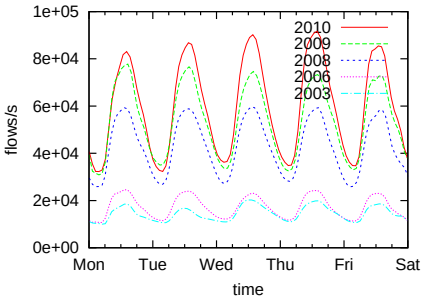


Figure 2.2: *The flow data measurement setup.*

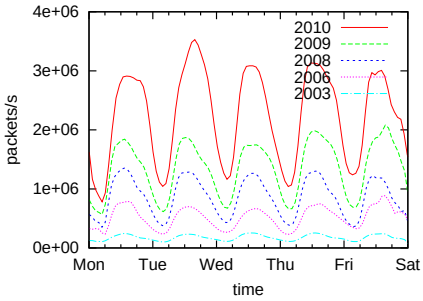
To characterize high-level traffic patterns, we extract from our data archives one 5-day trace (Monday, 00:00 UTC until Friday, 24:00 UTC) for every year between 2003 and 2010. We always choose the first full working week of November, which is in the middle of the fall term. Note that we had to exclude 5 hours from our eight traces due to router reboots, collector outages, or storage failures. Yet, this is negligible given an overall time of 960 hours covered by our traces. Figure 2.3 displays the observed number of flows, packets, and bytes per second. The large-scale nature of our measurements is underlined by high peak rates of more than 80,000 flows per second, 3 million packets per second, and more than 20 Gbit/s of traffic. In addition, the traffic characteristics show as expected strong daily patterns [145] and a steady traffic increase between the years 2003 and 2010.

This specific data set is used in Chapter 5 to characterize the long-term shifts within the Internet service landscape. However, within this work we use other time periods to analyze for example specific network outages or correlate the flow-level data with log-level or packet-level information. Note that those traces will be introduced at the point of their first usage including a description of the applied data preprocessing.

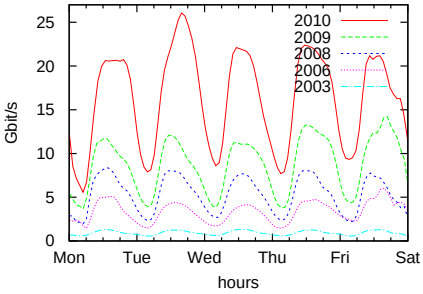
external links, 2008-2009: 14 external links.



(a) Observed flows.



(b) Observed packets.



(c) Observed bytes.

Figure 2.3: *5-day flow traces between 2003 and 2010: observed number of flows, packets, and bytes per second.*

2.2 DPI Application Labels

We deploy a commercial DPI appliance at the uplink of the ETH Zurich (ETHZ) network. As illustrated in Figure 2.4, traffic crossing the uplink of ETH Zurich and leaving the SWITCH network toward the Internet (or flowing the opposite direction) is recorded by the DPI appliance and the flow sensors at the border. To cope with peak rates of 2.5 Gbps, the DPI appliance is equipped with 16 CPU cores, 24 GB of RAM, and a dedicated capturing card. This allow us to combine both measurement data streams, and enrich the flow-level data with packet-level data. This simplifies the development of new algorithms or allows the validation of flow-based approaches.

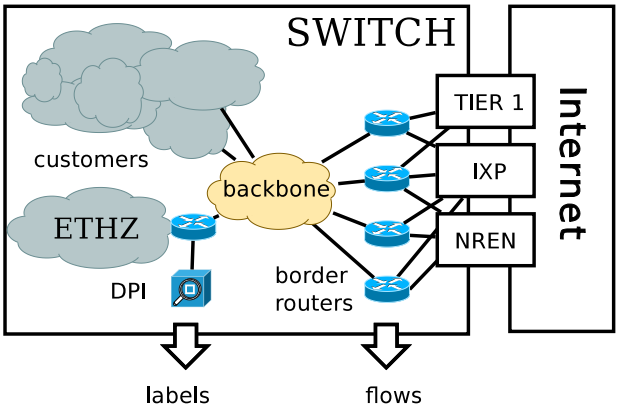


Figure 2.4: *The DPI measurement setup.*

The DPI appliance is capable of identifying more than 500 network applications and allows the user to access the corresponding meta or payload information. We like to point out that the term network application in the sense used by this appliance is not limited to network applications that are standardized by Internet Engineering Task Force (IETF). For example, the appliance identifies Skype traffic or classifies HTTP based applications into finer categories such as 'google chat', 'facebook', or 'doubleclick ads'. As soon as a specific network application is identified, the corresponding meta information can be exported. For example, the DPI appliance can be configured to export all login names, passwords, and email addresses that are exchanged over the 'facebook' application.

Since such payload-related information is highly privacy-sensitive and is not required for our research, we configured the DPI appliance to export only application identifiers without any packet payload. In more details, the DPI appliance reports only the most specific application that is identified within a given stream. For example, if a flow is used to deliver an advertisement of Doubleclick over HTTP/TCP/IP this flow is reported only as 'doubleclick ads'. However, if the application above the TCP stack is not identified, then the flow will be labeled as 'tcp'. In addition to the application identifier a flag is exported that indicates which of the communication end points is the "server side" of the flow. A sample of the exported data is provided in Appendix A.

To shortly characterize the high-level traffic patterns observed by the DPI appliance, we present the traffic mix by data captured at July 4 2011. While one day of flow-level data contains some 3.4 billion flows, a respective data set from DPI appliance includes more than 660 million application tuples, amounting to 55 GB of binary data. 92.3 % of the application tuples are based on TCP or UDP. Table 2.1 provides an overview of the top 20 application types our DPI appliance detects. Possibly surprising is the fact that NTP and Skype traffic dominate due to the fact that they generate a large number of small connections or flows. The high number of NTP labels can be explained by the fact that ETH Zurich (ETHZ) hosts a popular NTP server.

This specific data set is used in Chapter 7 to develop new types of service classification schemes. However, within this work we use other time periods to validate our proposed flow-based methods. Note that those traces will be introduced at the point of their first usage including a description of the applied data preprocessing.

2.3 HTTPS Services Labels

As long as the data stream is not encrypted, DPI appliances are an helpful information source to extract application labels. However, for services running over HTTPS, the DPI appliance can only inspect the content of a TLS/SSL handshake to predict the application. Since this handshake is standardized and therefore similar for most applications, this traffic is often labeled with the tag 'https'. Therefore, to develop and validate our HTTPS specific service annotation method discussed in Chapter 6.1, we were required to manually label a data set.

top	application	% of labels	top	application	% of labels
1	ntp	29.1	11	google	0.5
2	skype	27.7	12	imaps	0.5
3	dns	12.0	13	smtp	0.4
4	http	10.9	14	unknown	0.3
5	tcp	5.3	15	rtp	0.3
6	https	4.2	16	rtcp	0.2
7	udp	3.5	17	smb	0.1
8	bittorrent	1.6	18	pop3s	0.1
9	ssh	1.4	19	google_ads	0.1
10	facebook	0.6	20	established	0.1

Table 2.1: *Top 20 application labels within the ETHZ network (July 4, 2011).*

To build this data set, we first extract from a 1-hour trace recorded on Monday noon 2010-03-15 the 500 most popular internal HTTPS hosts/sockets (`top500`) in terms of unique IP addresses they which they communicate. Then, we manually access each of the 500 obtained IP addresses with a web browser and determine the host type (e.g., Horde, OWA). In cases where this does not work (e.g., Skype), we use `nmap` to actively probe the service. Table 2.2 summarizes the results of this time-consuming task.

class	type	# servers	# flows (mil.)
mail		77 (15.4%)	362.4 (66.0%)
	OWA	52	172.1
	Horde	10	84.9
	others	15	105.4
non mail		398 (79.6%)	159.0 (29.0%)
	WWW	137	88.9
	Skype	153	45.9
	VPN	15	8.3
	other	93	15.9
unknown		25 (5%)	27.7 (5.0%)
total		500	549.1

Table 2.2: *`top500` HTTPS services within the SWITCH network (March 15, 2010) including the number of exchanged flows (March 14-25 2010).*

We would like to point out the difference in the methodology between the collected HTTPS Service Label compared to Application Label collected by

the DPI appliance presented before. In this section, we classified the connection endpoint, more precisely the SeSs listening on TCP port 443, in different types such as Outlook Web Access (OWA), Skype, or VPN using manual investigation.

2.4 Mail Server Log

To better understand the traffic patterns caused by modern mail servers and to develop and validate our mail related troubleshoot application discussed in Chapter 8, we use a log of a university mail server serving around 2,400 user accounts and receiving on average 2,800 SMTP flows per hour. The analysis of the log unveiled that as much as 78.3% of the SMTP sessions were rejected during an early state of the SMTP session establishment. SMTP sessions that are early rejected end up with only a few exchanged bytes, a property that is discussed in more detail in Chapter 8.

Chapter 3

FlowBox: A Toolbox for On-Line Flow Processing

The processing of voluminous flow measurement data is a challenging task since it requires to process millions of flows within limited time to deliver statistics, reports, or on-line alerts to the user.

In this chapter we present FlowBox, a toolbox for on-line flow processing that addresses the needs of network operators and researchers. The core of FlowBox is designed to allow for multiple processors and parallel processing to speed up data mining. To reduce the time required to build a new analysis or alerting system, FlowBox is organized in loosely coupled modules that can easily be reused or extended to implement the required functionality within short time. Additionally, FlowBox provides an interface written in Ruby, which allows users to quickly evaluate new ideas in a comfortable way.

In the thesis at hand, we describe the FlowBox units that can be used for analysis and for troubleshoot applications such as the server socket extractor discussed in Chapter 5 or the remote connectivity issue tracker presented in Chapter 9.

The rest of this chapter is structured as followed: Section 3.1 describes the design principals used in FlowBox. Section 3.2 provides an overview of the implementation and evaluates the performance study of FlowBox. Section 3.3 presents the related work. The summary follows in Section 3.4

3.1 Design

The design of a flow processing tool should address two main requirements: throughput and flexibility. In the next section we present our design of FlowBox that exploits concurrency to improve the overall processing throughput. Afterwards, we present our layered pipe architecture that combines the advantages of different programming languages to satisfy the flexibility requirement.

3.1.1 Pipeline Design Pattern

To take full advantage of modern multi core architectures, FlowBox relies on the pipeline design pattern [93,154] that is often used in programming to split the task into multiple subparts and process them in parallel. As illustrated in Figure 3.1, the incoming flows are grouped into containers and forwarded along the pipeline to multiple worker units. Note that different units process the flow containers simultaneously. As soon as the work is completed, the flow container is forwarded to the next unit. To keep the units loosely coupled [1], the flow containers are stored into a shared queue that is accessible by the interconnected units.

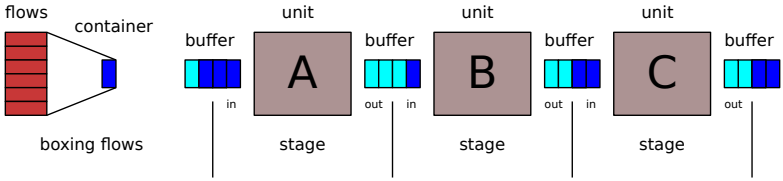


Figure 3.1: *The pipeline desgin of FlowBox.*

A simple linear pipeline as illustrated in Figure 3.1 allows to distribute the processing load over n processes, where n is the number of units of the chain. In situations where processing load is unbalanced along the units (bottleneck), a non linear chain configuration could be used to improve the overall processing speed. For example, to speed up the calculation of cryptographic flow hash keys this specific stage of the pipeline could be run in parallel on the same host.

[1] Minimizing knowledge of each unit about all other units is required to achieve an extensible software design [103].

Besides the duplication of certain parts, the design allows to instantiate certain pipe stages on new hosts and forward the data over the network. Hence, we can combine the resources of multiple hosts and parallelize and speed up the flow processing as illustrated in Figure 3.2. This interesting feature allows FlowBox to combine the resources of multiple off-the-shelf servers of a data center to build a distributed flow processing chain.

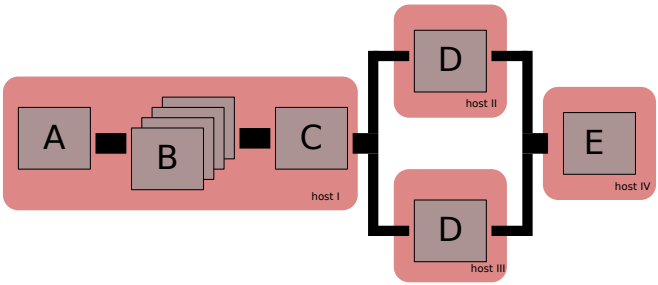
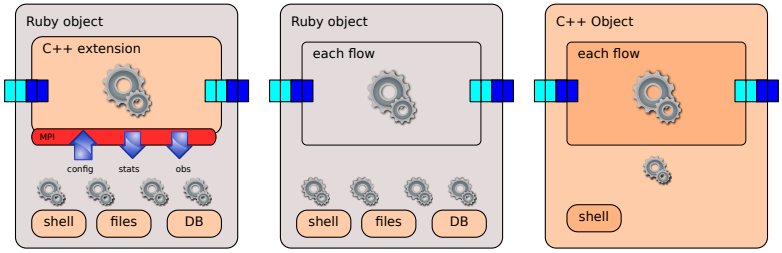


Figure 3.2: *A non linear FlowBox pipeline across multiple hosts.*

3.1.2 Layered Architecture

Extending and testing an application written in C/C++ requires to recompile it. Depending on the available system performance and the used build chain, this is a time consuming and inconvenient task to develop new ideas! To simplify this work flow and provide higher flexibility for rapid prototyping without sacrificing throughout we propose to use a layered modular architecture in FlowBox.

In more detail, we combine the compiled languages C++ and the object oriented scripting languages Ruby to build a layered unit design. In fact, we exploit that Ruby objects can be extended by functions that are written in C/C++. This allows to move computational and/or I/O intensive tasks into a C++ extension, while the rest of the functionality is written in Ruby as illustrated in Figure 3.3(a). This has the effect that flow data is mainly processed by the C++ extension. Surrounding tasks such as configuring the unit or post-processing of statistics and observations is implemented in Ruby. This encapsulation approach allows to implement the main routine as a simple Ruby script that configures and arranges a set of units to a processing chain. The script can easily be changed and modified without recompiling the different



(a) Standard unit (b) Development unit (c) C++ unit

Figure 3.3: *The layered architecture provided by FlowBox to implement different worker units.*

units. Even users with limited programming skills can thus configure flow processing chains in a short time.

To speed up the development of new ideas, this layered approach enables users to even build units where the flow processing is solely based on Ruby as illustrated on Figure 3.3(b). Then, developers can access and modify the content of the flows using simple statements written in Ruby. Of course the throughput is reduced since the script needs to be interpreted by the Ruby interpreter at runtime. Nevertheless, the performance impact can be limited by building mixed chains, combining both types of modules together at the same time. This implies that standard tasks such as reading flows from disk or filtering duplicated flows are implemented by units using a C++ core.

In addition, we designed the API of the core flow processing routines in a way that allows us to build pure C++ based processing chains. This is valuable in case a company/user is required for e.g. security, maintenance, or coding policies to run the flow analysis application as compiled executable where no code is interpreted. Moreover, this option is suited for well-tested and rather static analysis components that are unlikely to change. In addition, we us this option to assess the performance impact of the Ruby interpreter on the flow processing in the evaluation section.

3.2 Implementation

In this section we provide an overview of the implementation of FlowBox. Furthermore, we discuss the throughput performance achieved by FlowBox.

More details about the use of FlowBox including a sample program illustrating the use of the rapid prototyping interface can be found in Appendix B.

3.2.1 Architecture

The FlowBox project consists of three major sub-projects: FlowBoxCC, FlowBoxRuby, and FlowBoxRails. All of them are released under the GNU Lesser General Public License. FlowBoxCC consists of multiple C++ classes and implements the core flow processing routines. For example, this library includes classes to efficiently parse NetFlow 5 or 9, to filter duplicated flows, to block flows based on IP blacklists, or to determine the autonomous system number of an IP address. These classes are designed to be thread-safe by protecting shared data structures with mutexes or semaphores where required. The FlowBoxRuby subproject contains the wrapper code around the core flow processing routines and includes pure Ruby classes to simplify the configuration of the different units. The last sub-project is FlowBoxRails and consists of major building blocks required to implement a web frontend for a FlowBox processing chain. It is based on the "Ruby on Rail" web framework and uses a database server to control the data processing in the background. FlowBoxRails provides a high-level user interface for, e.g., troubleshooting application that have to run in 24/7 manner over a long time period.

3.2.2 Evaluation

In this section we evaluate the performance of FlowBox by profiling an example application that counts the number of bytes that are exchanged over a network. We first discuss the setup used to collect the measurements and then discuss our findings.

Setup

To collect the performance measurements we use an off-the-shelf server equipped with four Quad-Core AMD Opteron 8350 HE processors, having 64GByte main memory, and running a Debian Linux 2.6.30. We compiled FlowBox using gcc 4.3.2 and and used Ruby Interpreter 1.9.2p290 to run the scripts. The flow throughput is estimated using two NetFlow V9 traces that span one hour and that were collected at *midnight* and *midday* 2nd of April, 2012 on the border of the SWITCH network (see Section 2.1). The traces contain 119.9

million and 330.6 million flows, respectively, as summarized in Table 3.1. The traces are compressed with bzip2 and stored on a local RAID 6 storage[2] requiring 4.3 GBytes. The decompression of the *midday* trace with the commandline tool `bzip2` requires more than 25 minutes and 16 GByte of disk space.

	midnight	*midday*
flows	119.9 M	330.6 M
file size	5.6G Byte	16.0 GByte
file size bz2	1.4G Byte	4.3 GByte
compression ratio	4.0	3.7
decompression time	438s	1,341 s

Table 3.1: *The traces used for the performance evaluation of FlowBox.*

Findings

To evaluate the performance penality imposed by the Ruby scripting language, we implement a byte counter unit in FlowBoxCC (in C++) and FlowBoxRuby (Ruby). The different units are assembled with a flow parser unit that reads the flows from compressed files.

case	real [s]	speedup	cpu [s]	load	flows/s
midnight_cc	223	16.1	441	197 %	538k
midnight_ruby	219	16.4	612	279 %	548k
midday_cc	789	4.6	1412	178 %	419k
midday_ruby	778	4.6	1875	241 %	424k

Table 3.2: *The performance evaluation of byte counter application.*

For each application we measure the running time (real) and the number of CPU-seconds used by the system on behalf of the process (cpu). Then, we calculate the percentage of CPU time that this job received (load). To eliminate noise introduced by background jobs, we perform 5 runs and always compute the median of the respective metrics, see Table 3.2.

We derive the number flows per second (flows/s) that are processed by the application by dividing the total number of flows by the running time. In

[2] Areca ARC-1220, 500GB, SATA II, 7'200 rpm

addition, relative speed measurement (speedup) is obtained by dividing the running time of the application by the trace duration, i.e., time spanned by the trace. Values of more than 1 indicate that the application can process data faster than real time, i.e., faster than 1 hour.

Based on our findings, we first point out that our byte counter application indeed is able to process the incoming flow stream of the SWITCH network in a online fashion. As Table 3.2 reveals, both implementations achieve a flow throughput of more than 400'000 flows per second. Medium-sized ISPs typically exhibit "only" peak rates of 50'000 flows per second. In the case of the large-sized SWITCH network our implementation still achieves the speedup factor of 4 during peak hours *midnight*. Without adding more parser units, our application can process four times more data. Interestingly, the number of processed flows per second is lower by almost 100k for *midday* compared to *midnight*. This can possibly be attributed to the significantly lower traffic volume and higher compression ratio of the bzip2 input files.

As discussed in Section 3.1.2, we expect that the Ruby-based counter sub-program *counter_ruby* is slower than its C++ counterpart since the code needs to be interpreted at runtime. As Table 3.2 reveals, *counter_ruby* requires 279 while *counter_cc* requires only 176 seconds. This shows that total load is increased by almost 30%. Nevertheless, the total running time is not affected, since this work was done in parallel with the reader. This illustrates the benefit of using independent worker units working in parallel.

In addition, we compared the byte counter application with a simple processing chain that only consists of a flow reader. The result is very similar. This indicates that most of the processing time is spent for reading flows. The question arises if our flow reader performs less well than expected. As Table 3.1 reveals, the pure decompression of *midnight* using bzip2 requires 438 seconds. In contrary, the systems requires only 220 seconds to execute *midnight_cc*, corresponding to a 60% reduced running time. This excellent performance of FlowBox is achieved by parallelizing the decompression using multiple threads. In addition, the system spends only 436 CPU-seconds for the *reader_cc*. This is equivalent to the time that the system spends to execute the decompression and illustrates that the actual parsing of the flows requires only a few CPU cycles compared to the decompression.

Overall, we believe that it does not make sense to further optimize the reader component for the following reasons. First, further parallelization of the decompression will increase the load on the general I/0 bus including disk and main memory. This can result in a decrease in total throughput of the

system due to cache thrashing. Yet, as long as no SSD storage is used, all improving efforts will be constrained by disk I/O at in the first place. Second, in operational networks flow meters generally export uncompressed measurement data over the network. Hence, such disk I/O or compression limitations do not exist. Instead, we propose to distribute the reader components over different physical hosts in a data center and to run multiple processing chains in parallel to scale Flow Box as discussed in Section3.1.1.

An additional evaluation focusing on the performance of the core data structures of FlowBox is presented in Appendix B. There, we try to mask possible limitations caused by I/O devices such as reading files from compressed files or network capturing. The findings indicate that the core of FlowBox is able to process up to 93.5 million flows per seconds if file I/O was ignored.

3.3 Related Work

Over the years, a diverse set of flow processing tools were developed. A wide set of products such as *IsarFlow* [64], *Arbor Peakflow* [4], *Lancope StealthWatch* [82], *Cisco Multi Netflow Collector* [26] can be used to collect and aggregate flow data for reporting or offline analysis. Extending those tools is difficult since source code is not publicly available. Other open source system such as *ntop* [35], *nfdump/nfsen* [57], *silk* [22] focus on interval statistics or user driven ad-hoc queries for root cause analysis. Closest to our work is the recently presented flow processing tool implemented in Java presented by Koegel et al. [76]. Both projects share the pipeline architecture suitable for multi-core stream based data processing. However, FlowBox provides the users more flexibility for rapid prototyping by combining complied and interpreted programming languages.

3.4 Summary

We presented FlowBox, a toolbox for on-line flow processing. FlowBox allows to process millions of flows within limited time to deliver statistics, reports, or alerts in a on-line fashion to network administrators. FlowBox exploits concurrency to improve the overall processing throughput and speeds up the data mining. The Toolbox is organized in loosely coupled modules that can easily be reused or extended to implement the required functionality within little time. In addition, FlowBox provides an interface written in Ruby,

a scripting language that is excellently suitable for rapid prototyping, allowing users to evaluate new ideas in a comfortable way. Our prototype application implemented in Ruby was able to process up to 500k flows per second on off-the-shelf hardware and achieved throughput rates of 93.5m flows per second if file I/O was ignored.

Chapter 4

Identifying Server Sockets

In this chapter, we introduce an application-agnostic method that uses flow-level measurement data to identify Server Sockets (SeSs), communication endpoints that offer a specific network service to multiple clients (such as webmail, Skype, DNS). This allows us to bundle flows belonging to the same network service into sets and study their characteristics. For example, we use this methodology in Chapter 5 to characterize the service landscape of the Internet or in Chapter 7 to scale up service classification results.

To keep our identification method application-agnostic we do not rely on port heuristics such as well-known ports and do not use protocol-specific information such as TCP flags. Identifying SeSs by analyzing the timing of the initial handshake between the communication endpoints is challenging due to the limited timing precision provided by flow-level data. Therefore the proposed heuristic analyzes only the communication patterns between the different endpoints to identify those that act as communication hub.

Our FlowBox based prototype uses a cache to remember already identified SeSs. This allows us to skip the data mining for flows belonging to already identified SeSs. Thanks to this caching strategy and by further eliminating irrelevant or duplicate flows, the data volume to be processed is massively reduced. This improves the overall scalability and makes it possible to process flow-level measurement data collected in large networks in an on-line manner such as the data set collected at the border of the SWITCH network.

4.1 Introduction

Most likely, there is no system with a similar degree of diversity in its usages as the Internet. The history of the Internet is characterized by the emergence of new and fast growing network services. Online social networks [43], video streaming [54], communication services (e.g., Skype, IRC), AJAX-based text processing, spreadsheet, or webmail applications [52] are just some examples. Besides these long-term changes, the popularity of each service instance can rapidly change within a few hours (flash crowds, viral effects). In this regard, the Internet is like a chameleon that is constantly changing its color.

In this chapter, we tackle the fundamental problem of identifying communication endpoints or as defined in chapter 1 SeSs. These SeSs are often used as basic building blocks in more advanced network monitoring applications [67, 71, 125]. Yet, to the best of our knowledge, only limited work is done in the field of identifying these data structures using flow-level data.

To close this gap, we contribute in this chapter an application-agnostic heuristic to identify SeSs from flow-level measurement data. Our approach is application-agnostic since we do not rely on port heuristics such as well known ports and do not use protocol specific information such as TCP flags to identify the service provider endpoint of a connection. Further, as we discussed in [147], flow-level data provides only limited timing precision. Therefore it is challenging to use the timestamps to distinguish the client from the service provider during the connection handshake.

Instead of relying on imprecise timing information, we propose an iterative greedy heuristic that analyzes the communication patterns between different endpoints to identify the SeSs. More precisely, we first aggregate the flows observed over a given time interval per communication endpoint. Then this communication graph is used to identify the endpoint that acts as biggest communication hub by which we mean an endpoint that communicates with the highest number of other endpoints. This node (endpoint) is then marked as a SeS and all its edges (interactions with other endpoints) are removed from the communication graph. After that, the next SeS is identified based on the updated communication graph. This procedure is repeated until all communication hubs are identified. Importantly, this iterative procedure also identifies SeSs that run on high ports or are involved in the exchange of only few packets and are hence frequently overlooked.

We implemented the heuristic using FlowBox, the flow processing toolbox presented in Chapter 3 and evaluated its performance using flow-level

measurement data collected at the border of the SWITCH network. To achieve a higher flow processing throughput we reduced the data volume by excluding irrelevant flows. For example, we use a cache that remembers the already identified SeSs. This allowes us to exclude flows that belong to already identified SeSs from further data mining steps. In addition, as reported by Wustrow et al. [159], real world network traces are full of background radiation caused by misconfigured or malicious hosts scanning the Internet. To cope with this effect, we introduce additional filtering steps to exclude this scanning traffic from further processing. Moreover, we eliminate duplicate flows that are due to the fact that some flows are visible at two vantage points of our measurement infrastructure. In our study, this filtering steps reduce the amount of flow records by one third, which significantly increase the performance of the general data processing. Furthermore, we use a set of validation checks that systematically exclude corrupt data from further processing right at the beginning.

In summary, we contribute an application-agnostic methodology to identify SeSs e.g., communication endpoints that offer a specific network service to multiple clients using flow-level measurement data. In contrast to existing work, our approach takes into account both TCP and UDP services, does not require TCP/SYN flags as used by Bartlett et al. [9], copes with noise from scanning [2], and with low resolution of NetFlow timestamps [147]. We find that our proposed techniques allow us to process a 5-day trace from the SWITCH network from 2010 in less than 4 days, suggesting that processing in real time is feasible.

The rest of this chapter is structured as follows: Section 4.2 provides a general overview of our approach. Section 4.3 explains the individual steps of our data processing including different filtering steps. Finally, we review related work in Section 4.4 and summarize our work in Section 4.5.

4.2 General approach

To identify network services in the wild, we need to take a coarser view of the data. In this section, we first discuss how these communication endpoints can be identified using flow level features and evaluate our proposal in Section 4.2.3. In Section 4.3 we will rely on this approach to efficiently identify and track network services.

4.2.1 Exploitable Information

Unfortunately, flow-based measurement data does not provide a field that indicates the SeS of a flow. Therefore, to identify them, we need to find a heuristic making an educated guess. In this section we discuss what properties across multiple flows can be exploited for this purpose.

To address this issue Bartlett et al. proposed in [9] to analyze the ”three-way handshake” of TCP connections to identify the service provider. Since TCP flags are exported by flow meters only in an aggregated fashion, this approach can not be applied in the context of flow-level data. In addition, certain flow meters are not even capable of exporting these flags at all [74]. Furthermore, such flags are restricted to TCP traffic and are not available for UDP. An alternative approach is to study which host started the communication (connection handshake) by analyzing the timing information provided by the flow data. Unfortunately, timing information provided by flow meters is unreliable and, therefore, cannot be used [147]. Finally, we could rely on the assumption that services are only provided on well known port numbers or low port numbers as proposed by [9]. Yet, many applications such as Skype or Bitttorent use software that is listen on high TCP and UDP port numbers to provide their service to other clients [146].

To overcome these issues, we propose a different approach: a service is likely to be contacted by multiple clients over time. Therefore we can observe many flows that start at diverse client sockets and end up at the same server socket. This effect will be visible in the flow traces as a “concentrator” or “hub” pattern. Evidently, such information can only be retrieved if sets of flows are collected and analyzed over time. Accordingly we propose to identify service sockets by analyzing the communication graph between the sockets by *aggregating flows over time and sockets* and study concentrator patterns in the communication graph.

4.2.2 Heuristic to Identify Server Sockets

In our notion of the communication graph, nodes correspond to sockets and edges correspond to flows between sockets. Our goal is to identify a subset of nodes that act as SeSs and are characterized by acting as communication hubs between endpoints (concentrator pattern). In principle, different strategies can be applied to extract these nodes from the communication graph, which is likely to result in different sets of identified SeSs. For example, we could randomly select a socket having a degree larger than 2 and add this socket

to the set of known server sockets[1]. After removing all edges of this server socket, the process is repeated until no nodes remain. Another strategy would be to always select the node having the highest degree and add such a socket to the set of SeSs.

Our choice is to minimize the number of SeSs required to describe the communication graph. This results in a compact description of the data improving the speed of further processing steps. Given this constraint, we assign each flow to a SeS and minimize at the same time the total number of SeSs. This problem can be transformed into a standard minimal vertex coverage problem. This problem is known to be NP hard [32]. Therefore, we propose to use the following greedy heuristic to approximate the solution: First, we build the communication graph. Then we select the node with the highest degree, add it to the set of identified SeSs. Next, we remove all edges that are incident to this SeS nodes. We repeat this until all nodes have a degree of less than a certain threshold. Intuitively, the obtained set of SeSs will include those services that are important and accessed by many users.

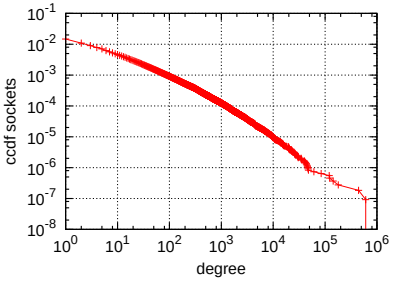
4.2.3 Analysis of concentrators

Now, we apply the greedy heuristic on a two hour trace that was collected from Wednesday, June 8, 2011 (10:30 UTC until 12:30 UTC) on the border of the SWTICH network (see Chapter 2). More precisely, we partition the trace into time intervals of 15 minutes and build the communication graph of the obtained flow sets. Then, we process the graph relying on our greedy heuristic.

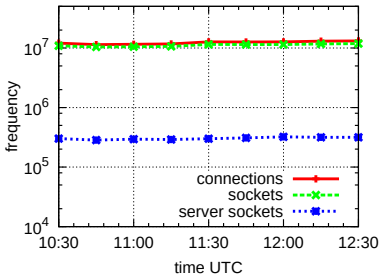
One important finding is that several of the top ranked SeSs belong to the peer to peer network of Skype. This is due to the fact that our heuristic is applicable even for P2P based applications, which may use high TCP/UDP port numbers. Further, we like to point out that the top ranked services are not necessarily those services that cause the most traffic in the network. As example, Skype super nodes are accessed by large set of peers causing a high degree in the communication graph. However the exchanged data volume is rather limited. We analyze this characteristic in more detail in Chapter 5.

We first count for each socket the number of incident edges found in the communication graph. This node degree is visualized in Figure 4.1(a). We find that around 90% of all sockets have only talked with a single socket. However, there exist other sockets that have exchanged data with more than

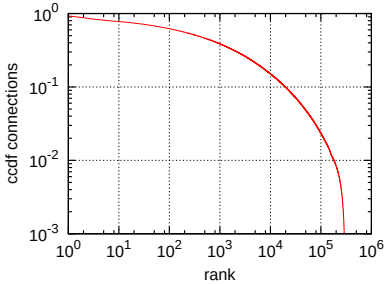
[1] Note that a concentrator pattern requires at least a node degree of 2



(a) Node degree of sockets.



(b) Number of connections, sockets, and server sockets.



(c) Percentage of connections covered by top x server sockets.

Figure 4.1: *The analysis of the communication graph.*

10,000 distinct sockets during the studied time interval of 15 minutes. These sockets are likely to correspond to popular network services accessed by many users, e.g., an important web server. Overall, this "binary" study of communication ("Has anyone talked with anyone?") already provides evidence for the existence of concentrators.

Figure 4.1(b) displays the number of connections and sockets that are observed in the communication graph during each 15-minute time interval. In addition, it reports the number of server sockets that were extracted by the greedy heuristic. We immediately notice that the number of connections averages around 11 million for each time interval. Surprisingly, the number of sockets (nodes) is not much smaller than the number of connections (edges). This effect can be explained as follows: If both sockets of each connection would be chosen randomly, then the number of observed sockets tuples is around twice the number of connections. However, the finding that the number of observed unique sockets is much smaller confirms the idea that one part of the connection is attached to a more stable endpoint, namely the server socket.

When extracting the SeSs using the greedy approach, we can drastically reduce the number of sockets: we receive less than 300,000 SeSs with more than one remaining incident edge instead of more than 10 million sockets. In principle, SeSs appear as the natural choice provided that they also explain a major share of all observed connections.

To this end, the x-axis in Figure 4.1(c) sorts sockets by the number of adjacent nodes in the communication graph. It shows in a CCDF curve the percentage of connections that are covered by the top x sockets. We observe that the top 20,000 SeSs out of 300,000 server sockets are involved in 90% of all connections during a typical 15 interval. Network professionals may decide against monitoring low-ranked SeSs if they are accessed only by few users or connections. Overall, the results of Figure 4.1(c) confirm that there indeed exist concentrators. This indicates that it can be feasible to keep state about already observed heavy concentrators and to monitor them across long time periods.

We also study the network applications that are offered by the top 100 concentrators or SeSs. To this end, we rely on information provided by a DPI appliance as introduced in Section 2. Since application labels are only available for parts of the SWITCH network, we cannot identify the type of service for 43 out of the 100 top SeSs, see Figure 4.2. Among the remaining SeSs, we find HTTP(S), DNS, Flash Real Time Messaging Protocol (RTMP), NTP,

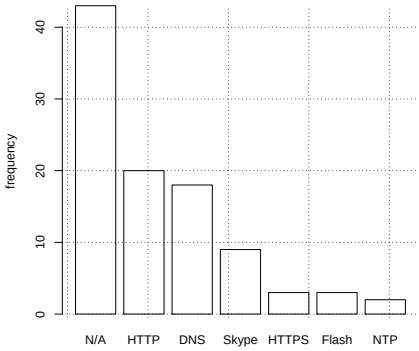


Figure 4.2: *The top 100 concentrator.*

and Skype. While the dominance of HTTP, NTP or DNS is expected, there are surprisingly also 7 Skype SeSs in the top 100 list[2]. In contrast to applications such as DNS or NTP, Skype services (probably supernodes) are likely to be less persistent, run on end-user machines, and adopt a P2P-like communication paradigm. It is one particular strength of our approach to highlight also such service types.

Applying the greedy approximation to extract SeSs turns out to be time-consuming. Computing the communication graph and extracting the server sockets for the two hours trace requires almost one day on a 2.2 GHz AMD Opteron 275 with 16GB memory. This underlines the need for recalling already detected SeSs as described in the next section.

4.3 Stream-based Implementation

Now, we describe how to implement our heuristic from the preceding section to process the measurement data produced by large networks in a on-line manner.

More precisely, our implementation includes seven major components as

[2]ETH Zurich is hosting a popular NTP server

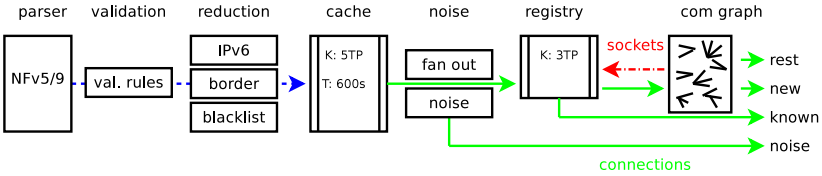


Figure 4.3: *An overview of the implementation.*

illustrated in Figure 4.3. The *parser* obtains unidirectional flow data from the central data repository described in Section 2.1. It decompresses bzip2 files and parses NetFlow version 5 or 9. Afterwards, a *validation* step is used to systematically exclude corrupt data from further processing. Then, a *reduction* step removes flows from the data stream that are not needed for the detection of SeSs. For a limited time, we keep the remaining flows in the connection *cache*. If we find for a unidirectional flow a corresponding reverse flow, we merge both into a connection and store them together in the connection cache. Any entry in this cache is uniquely identified by the IP address, protocol number, and port number of both the internal and the external host.

Up to this point, data processing is tickless, i.e., information about incoming flows is continuously inserted into the connection cache. However, further processing is clock-based, since the detection of SeSs requires studying a large set of flows across many connections within a certain time interval, see Section 4.2.1. Whenever a certain time interval has elapsed, we extract all connections from the connection cache that have been observed during that time interval. Then, we perform an additional step of filtering on these flows, mitigating the impact of scanning activities and eliminating general *noise*.

Finally, we identify from the remaining set of connections the SeSs, i.e., the side of a connection that actually offers the "service". To this end, our data processing first does a lookup in the *registry*. This component recalls already identified SeSs from previous time intervals, which are identified by the three-tuple of IP address, port, and protocol number. For connections after the filtering step we distinguish between two cases: if there is already an entry for one side of the connection in the registry, the server socket is already known and we are done. Otherwise, we forward the connection to the *socket detection* engine. There, we adopt a greedy heuristic as outlined in Section 4.2.2 to identify SeSs. After the detection of new SeSs, we add them to the registry.

In the following, we provide more details about the implementation of

the individual components. Section 4.3.1 outlines our correction methods applied to exclude measurement errors. Section 4.3.2 summarizes our filtering to reduce the data volume. Section 4.3.3 explains our data structure for the connection cache and in Section 4.3.4 we discuss the processing used to mitigate the impact of noise and scanning. Further, we introduce our heuristics to detect new SeSs in Section 4.3.5. Finally, Section 4.3.6 discusses the performance of our system.

4.3.1 Validation

Cleaning of input data is broadly accepted as good practice for application programming. What is considered to be good practice for general programming should also apply measurements applications. Therefore, in this section, we shortly discuss the data sensitization methods applied to our flow data to avoid any side effects to our experiments

We examine the various error sources based on a generic flow measurement setup [121]. We found that most of the errors caused by the metering process can be identified using rules that include additional knowledge about the observed network such timeout setting of the flow meters, max. link speed of the network, or max. MTU. This allows us to identify flows that can be caused by clients attached to the network and are therefore caused by an artifact of the metering system.

A subset of the rules are listed in Table 4.1. For example, the first rule exploits the fact that flow meters export long-lived flows after a time duration of $active\_timeout$ (typically 5min). Therefore a valid flow should start and end within a sliding windows of size $2 \cdot active\_timeout$ around the current time. If a flow violates this restriction, we report the case for further debugging and exclude the flow from further processing.

4.3.2 Reduction

Altogether, we perform three reduction steps: first, we remove IPv6 flows. In 2011, the traffic volume of IPv6 is still far below 10%. Still, we point out that our techniques can be directly applied to IPv6 data without any major modifications. Second, we only keep those flows that either originate or terminate inside our network. In particular, traversing flows or network-internal flows are removed. By doing so, we ensure that if traffic is bidirectional between a pair of hosts, we always must observe both directions. Finally, we filter out a

1	Time freshness	Flow timestamps are within a sliding window of size $2 \cdot active\_timeout$.
2	Time causality	Flow end time is larger than flow start time.
3	Min byte size	Byte counter is $\geq 20/28/40$.
4	Bitrate	Bytes / flow duration is $< max\_bitrate$.
5	Min packet size	Packet counter is > 0.
6	Packetrate	Packet / flow duration is $< max\_packetrate$
7	MTU	The ratio bytes divided by packets is $< MTU$ and $> 20/28/40$.

Table 4.1: *A subset of the used validation rules.*

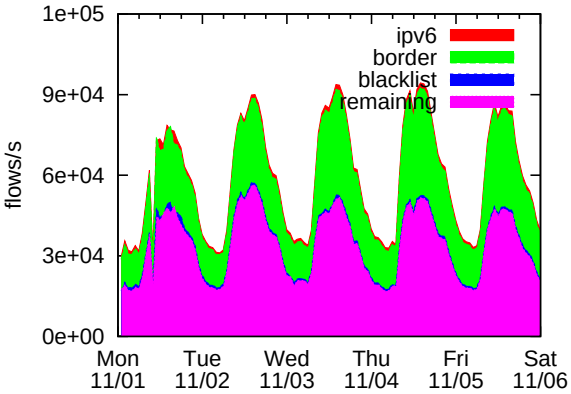


Figure 4.4: *The result of the data reduction steps.*

flow if its source or destination is on our manually generated prefix blacklist. For example, we ignore PlanetLab hosts and eliminate bogon IP addresses based on the bogon prefix list that we obtained for the year 2010 [139].

The plot of Figure 4.4 shows for a five day trace (first full working week of November 2010) the flow rates averaged over 15 minutes for IPv6 (*IPv6*), internal and traversing flows (*border*) and flows from or to hosts found on our *blacklist*. While the *IPv6* and *blacklist* filters only remove a limited amount of data, the *border* filter eliminates generally more than 40% of all flows. In total, there remain around 50% of all flows after this step.

4.3.3 Connection cache

After the preprocessing step, the unidirectional flows arrive at the connection cache. One central task of this component is to merge unidirectional into bidirectional flows if the source identifiers (IP address, port number) of a flow match the destination identifiers of another flow, and vice versa. After this step, we generally speak about *connections* irrespective of whether we found flows for both directions, or only for one direction. We maintain a hash-like data structure that, for observed connections identified by IP addresses, protocol number, and application ports, stores and updates information that is relevant for further analysis. This includes packet counts, byte counts, and the time when the connection was active (start and end time).

Data processing up to the connection cache is stream-based. Incoming flow information is continuously parsed, filtered, and finally inserted into the connection cache. Subsequent data processing is clock-based, because we need to consider a larger set of connections across a certain time interval t to detect SeSs, see Section 4.3.5. Therefore, we partition the timeline into intervals of t minutes, and proceed with our data processing whenever such a time interval has elapsed. For t we choose 15 minutes, which makes it likely to observe at least some flows for any active server socket. Every 15 minutes we extract from the cache connections that have been active during the preceding 15 minutes time interval, delete them from the cache, and pass them on to the filtering unit described in Section 4.3.4.

Efficient memory handling ensures that data does not have to be copied when "forwarding" connections from the connection cache to the subsequent units for noise elimination or SeS detection.

Figure 4.5 shows for our 2010 trace the number of connections that are stored in the connection cache. We find that the cache generally contains between 7 and 28 million connections. As expected, the curve follows diurnal patterns [145], but also shows spikes. We speculate that such rapid increases in the number of stored connections are due to scanning activities.

4.3.4 Noise Elimination

Whenever a time interval of 15 minutes has expired, flows that have been active during the preceding interval are removed from the cache and undergo an additional step of filtering, see Figure 4.3. Our goal is to eliminate noise that could impede the detection of SeSs.

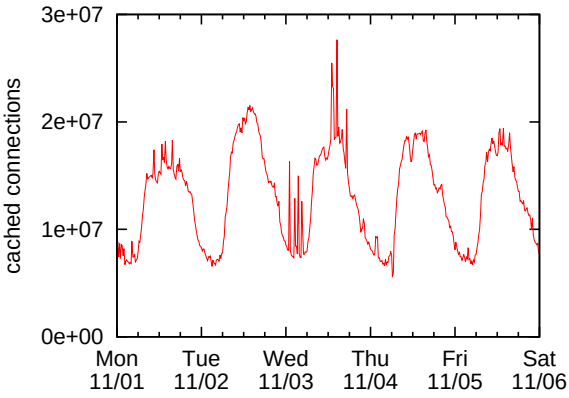


Figure 4.5: *The size of the connection cache .*

First, we try to cull potential scanning activities. To this end, we borrow the key idea from Allman et al. [2]. Scanning attempts generally do not result in established connections, and are not limited to individual hosts or ports. The pseudo code listing of Algorithm 1 summarizes our approach. We examine the fanout of hosts, i.e., the number of 2-tuples (IP address, port number) that a host tries to access. More precisely, we associate two counters with each host: fan_{good} reflects the number of bidirectional TCP (UDP) connections with more than 2 (1) packet(s) in both directions in which this host participates. Similarly, fan_{bad} represents the number of unidirectional flows where this host is listed as source. Following the result of Allman et al. [2], we finally classify a host as a scanner if it has fan_{bad} of at least 4 and if fan_{bad} is at least twice fan_{good}.

Connections that are associated with a detected scanner are removed from further data processing. Figure 4.6 shows that adopting this strategy reduces the incoming connections per second by 22% on average. More importantly, we find that many of the spikes disappear, indicating the efficiency of our scan detection heuristic.

We point out that the discussed heuristics do not eliminate connections with low traffic volume such as pure TCP 3-way handshakes. However, we are mainly interested in connections where a minimum amount of data is exchanged. Therefore, a second filtering unit eliminates all (i) unidirectional TCP connections, (ii) TCP connections with fewer than 4 packets, and (iii)

Algorithm 1 The scanning detection algorithm.

$fan_{good} = \{\}$
$fan_{bad} = \{\}$

for all connections $|c|$ of connection cache **do**
  if (bidirect. TCP **and** > 2 in/out pckts) **then**
    $fan_{good}[c.ipaddr\_in]++$
    $fan_{good}[c.ipaddr\_out]++$
  else if (bidirect. UDP) **then**
    $fan_{good}[c.ipaddr\_in]++$
    $fan_{good}[c.ipaddr\_out]++$
  else if > 0 out packets **and** 0 in packets **then**
    $fan_{bad}[c.ipaddr\_in]++$
  else if > 0 in packets **and** 0 out packets **then**
    $fan_{bad}[c.ipaddr\_out]++$
  end if
end for
for all IP addresses ip in the connection cache **do**
  if ($fan_{bad}[ip] > 4$ **and** $fan_{bad}[ip] > 2 \cdot fan_{good}[ip]$) **then**
    $\Rightarrow$ **SCANNING**
  end if
end for

UDP connections with fewer than one packet in both directions. As shown in Figure 4.6 ("noise"), this decreases the connection rate per second by $1,300$ on average.

4.3.5 Detection of Server Sockets

As shown in Figure 4.3 and discussed in Section 4.2, we implement a *registry* component that remembers already detected SeSs and passes only those connections to the detection engine that cannot been associated with any already known server socket. After a certain warm-up phase, the registry reduces the number of connections per second that need to be forwarded to the server detection unit by 95%. Evidently, keeping and recalling already detected SeSs significantly decreases the computational load for SeS detection.

The remainder of this section will introduce the heuristics that we use to

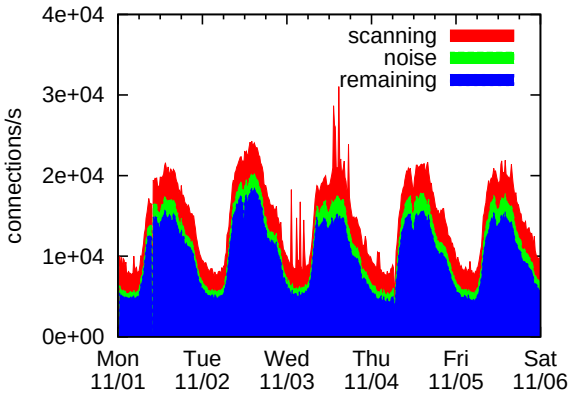


Figure 4.6: *The effect of noise elimination.*

detect new SeSs given a set of still unassigned connections.

Our approach is guided by two key ideas. First, we regard 3-tuples (IP address, port number, protocol number) as hot candidates for being a SeS if according to our connection data these sockets communicate with a large number of other endpoints (again, identified by IP address, port, protocol number). Second, we adopt a greedy approach, see Section 4.2.2 that first selects the 3-tuples as SeSs with the highest number of associated "client" 3-tuples. We iteratively remove first the "important" SeSs, delete all their associated connections, and then continue with the SeS detection based on the remaining set of connections.

The pseudo code listing of Algorithm 2 illustrates in a simplified way how our approach works. Initially, we compute two separate lists, one for all 3-tuples that are inside our studied network, and the other for all 3-tuples that are outside our network. Moreover, we sort both lists by the number of "client 3-tuples" with which a 3-tuple communicates (degree $deg(x)$). Within the main lookup, the first while loop starts extracting all 3-tuples from inside our network until the maximum number of "client 3-tuples" falls below the maximum number of "client 3-tuples" for the external candidates. Accordingly, the second inner loop extracts SeSs that are outside of our network. Every time after choosing a server socket, we remove all connections associated with the detected server socket, and recompute only the list SS_{in} or SS_{out}, respectively. Overall, the extraction process continues until all remain-

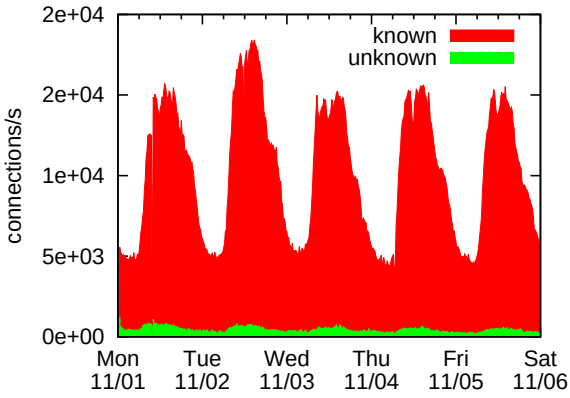


Figure 4.7: *The registry – known vs. unknown server sockets.*

Algorithm 2 The algorithm to detect server sockets.

compute list SS_{in} {int. sockets sorted by # ext. clients}
compute list SS_{out} {ext. sockets sorted by # int. clients}

while ($\deg(SS_{out}[0]) > 2$ **or** $\deg(SS_{in}[0]) > 2$) **do**
  while ($\deg(SS_{in}[0]) > \deg(SS_{out}[0])$) **do**
    $ss = SS_{in}[0]$ {classify ss as internal server socket}
    remove ss from SS_{in}
    update deg() for all entries of SS_{in}
  end while
  while ($\deg(SS_{out}[0]) \geq \deg(SS_{in}[0])$) **do**
    $ss = SS_{out}[0]$ {classify ss as external server socket}
    remove ss from SS_{out}
    update deg() for all entries of SS_{out}
  end while
end while

ing internal and external candidates have a degree of fewer to or equal than 2.

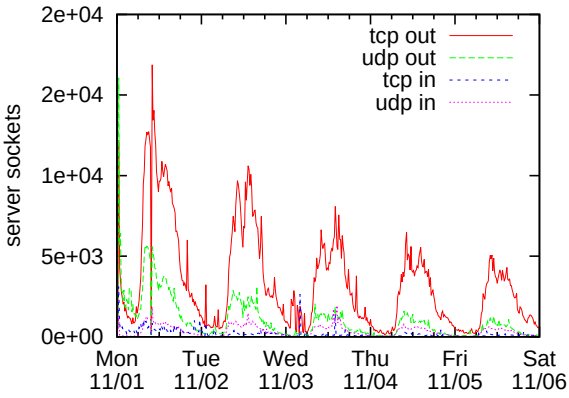


Figure 4.8: *The number of new identified server sockets.*

4.3.6 System Performance

Figure 4.7 reveals that 96% of all connections that enter the server detection component per second can be assigned to a server socket. Only 4% need to be classified by the server detection unit. By design (see Algorithm 2) the remaining connections are associated with 3-tuples that talk with fewer than three other "client" 3-tuples. In addition to the expected daily patterns in the connection rates, we see that the number of unclassified connections steadily decreases during the 5 days of our trace, meaning that more and more connections can already be assigned to a SeS by the registry.

In contrast to connection rates, Figure 4.8 displays the number of newly detected sockets per 15-minute time intervals across the runtime of our 5-day trace from 2010. In general, most new SeSs (peaks of 5,000 to more than 16,000) are located outside our studied network and use TCP, followed by external UDP sockets, internal UDP sockets, and internal TCP sockets. Generally, we find a higher number of new sockets in the beginning of our trace, since only few SeSs will be known to the registry in the warm-up phase.

So far, this section has illustrated the challenges of processing large-scale traffic data with peak rates of more than 20 Gbit/s. Therefore, we had to carefully design our data structures and implementation. Overall, we have implemented more than 12,000 lines of code. While all performance-critical parts of our code are in C++ , we sometimes also rely on Ruby to instrument our measurements and to profile our system. On a 2.2 GHz AMD Opteron 275

with 16GB memory[3], we can process our 5-day trace from 2010 within 3.8 days. Older traces are considerably faster, within the 2003 trace finishing with 0.8 days. The memory required by our approach is dominated by the number of connections in the connection cache (see Figure 4.5) and the data stored per connection. Typically, we need around 128 bytes per connection and 9 GB of memory in total.

These numbers clearly suggest that our approach can come close to real-time processing. This holds in particular if we do not have to use shared hardware resources as for this thesis. Therefore, we can picture exciting potentials of our measurement approach and the proposed data processing techniques if they are permanently applied to a real network. For example, they can provide network operators with an overview about network applications, their usages, and their location.

4.4 Related Work

Several proposals have been made to detect network services, many of them relying on active probing, e.g., [9, 85] instead of passively collected flow-level data as in this thesis. Another line of work has tried to compare the power of passive vs. active service discovery [9, 149]. Although detection of well-known services can be "straightforward" as stated by Bartlett et al. [9], our work clearly reveals the challenges (e.g., missing TCP flags and imprecise timer information) for a more large-scale data set, and, importantly, studies high-port applications. In contrast to a packet-level solution [149], collecting flow-level data can scale well for large ISPs with more than 2 million internal hosts. However, due to the limited nature of flow-level information, it becomes more challenging to process the data and to draw conclusions [109, 127].

Possibly closest to our work is AutoFocus [42] that uses clustering techniques to decide what the right granularity is to group conspicuous traffic sources together. Unfortunately, it relies on some implicit assumptions. For example, it allows to define clusters based on the value of port numbers or IP addresses, although similar or "close-by" port numbers and IP addresses do not necessarily reflect the same or similar applications (e.g., SSH on port 22 and SMTP on port 25).

[3] Machine was shared with other users.

4.5 Summary

The main objective of this chapter is to contribute application-agnostic, data-driven methodology to identify SeSs e.g., communication endpoints that offer a specific network service to multiple clients using flow-level measurement data. Our approach takes into account both TCP and UDP services, does not require TCP/SYN flags, copes with noise from scanning [2], and with low resolution of NetFlow timestamps. Further we demonstrated that our method is capable of processing flow level measurement data from large level data in real time.

Part II

Birds Eye View of Internet Services

Chapter 5

Characterizing the Service Landscape

In the last chapter we introduced the a method to identify SeSs that allows us to study the service landscape of large-scale networks using flow-level measurement data. In this chapter we apply this method to flow traces that were collected over a time span of nine years at a backbone of a national ISP. This study helps network operators and researchers to gain more insights about *where* network services are located, *what traffic characteristics* they show, and *how services changed over time*. For example, our findings reveal that the existence of end-user deployed services is more prevalent than widely thought. Further, a frequent item set analysis of the location of network services reveals that they are often co-located in the *address*×*port space*.

Besides improving the general understanding of how network resources are used, we believe that this study will foster new ideas in the area of flow-based service monitoring. In our case, this work lays the foundation for the service annotation methods that we discuss in Chapter 6 and the connectivity troubleshooting application of Chapter 9.

5.1 Introduction

The recent history of the Internet is characterized by the rapid rise and fall of the popularity of many of network services. Online social networks [43],

video streaming [54], AJAX-based text processing, spreadsheet, or webmail applications [52], are just some examples of the constantly changing service landscape of the Internet. While the main focus of previous studies [15, 81] has been to analyze trends on aggregate Internet traffic (e.g., "how much has P2P traffic increased or decreased in total?"), we shed light on the service landscape of the Internet at the granularity of individual Server Sockets (SeSs) using a large-scale data source.

In addition to aggregate traffic statistics, we present our results on the number of observed SeSs, the amount of traffic per SeSs, analyze their location, and the port ranges used. One particular strength of our approach is to identify end-user deployed network services (e.g., Skype or Bittorrent supernodes) that are frequently accessed but unfortunately widely overlooked by existing traffic studies. Furthermore, studying the SeSs across time we find that a significant share of services are available over longer time periods. For example, this can be leveraged to detect the advent of new popular services and potentially abnormal usage patterns. In addition, we study the spatial structure of SeSs, focusing on the SeSs that reside on the same host. We illustrate that the set of used transport-layer ports can show typical patterns, providing hints about the role of a host (e.g., a VPN server). This finding can pave the way for more sophisticated service classification techniques.

The rest of this Chapter is structured as follows. Section 5.2 gives an overview of used data sets and discusses the data processing applied to extract server sockets. In Section 5.3 we present the characterization of the service landscape. Section 5.4 summarizes related work before we conclude in Section 5.5.

5.2 Methodology

Unless otherwise noted, we report our findings based on an unsampled NetFlow V9 trace that was collected from Wednesday, June 8, 2011 (00:00 UTC until 24:00 UTC) at the border of the SWITCH network (see Chapter 2) further simple called `trace`. The large-scale nature of our measurements is underlined by peak rates of more than 80,000 flows per second, 3 million packets per second, and more than 10 Gbit/s of traffic. To study the service landscape captured by `trace`, we rely on the concept of SeSs as introduced in Chapter 4. More precisely, we use the implementation described in Section 4.3 to identify the SeSs.

In addition, to analyze the evolution of the service landscape presented in Section 5.3.7, we extract from our data archives one 5-day trace for every year between 2003 and 2010 (see Table 5.1) and apply the same methodology to study the SeSs. An overview about the number of observed flows, packets, and bytes per second captured by this data set is given in Section 2.1.

year	start	end	compressed size
2010	Mo 1. 11	Fr. 5. 11	322 GB
2009	Mo 2. 11	Fr. 6. 11	289 GB
2008	Mo 3. 11	Fr. 7. 11	247 GB
2007	Mo 5. 11	Fr. 9. 11	178 GB
2006	Mo 30. 10	Fr. 3. 11	107 GB
2005	Mo 31. 10	Fr. 4. 11	81 GB
2004	Mo 1. 11	Fr. 5. 11	97 GB
2003	Mo 3. 11	Fr. 7. 11	87 GB

Table 5.1: *The 5-day traces used to study the evolution of SeSs*

To complete our view of the network, we use a commercial DPI appliance at the single uplink of ETH Zurich as described in Section 2.2. The configuration of the DPI appliance provides an application ID for flow-style *tuples* identified by protocol number, src/dest port and IP addresses. The precision of the application ID varies, with the returned IDs representing for example TCP, HTTP, or Bittorrent. In this chapter, we ignore any labels that are less specific than TCP or UDP. By correlating information from `trace` with such *labelled tuples*, we can label a subset of the SeSs observed in `trace` and complete our view on the network.

5.3 Findings

In this Section we present our findings with respect to the characterization of the Internet service landscape based on the methodology discussed in Section 5.2. First, we study the fundamental characteristics such as the number of bytes, packets, and connections per SeS using the short data set `trace`. Then, we discuss the long term evolution of the service landscape in Section 5.3.7.

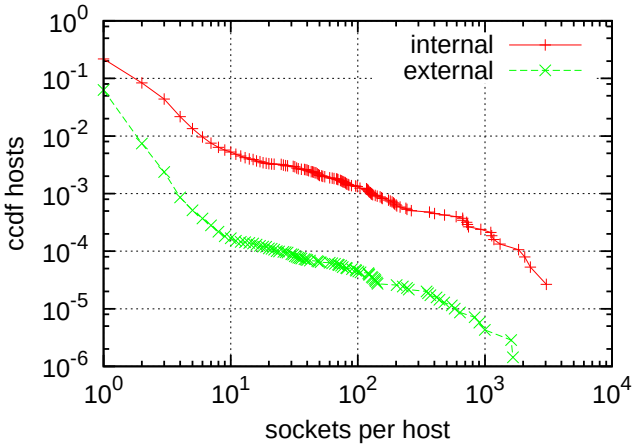


Figure 5.1: *The number of sockets per host.*

5.3.1 Server sockets per host

We start by studying how many of our detected SeSs are located on a single host,[1] see Figure 5.1. As shown, for our 40,000 internal and 700,000 external hosts, there is only a single active SeS for 78% of the internal and 93% of the external hosts. The difference between external and internal SeSs is expected, since we are likely to see only a part of all traffic reaching the external hosts. Yet, less than 1% of the internal and external hosts show more than 10 sockets during the 24 hours. Manually checking, we find that many of these hosts are FTP servers (with port 21) that rely on passive mode to exchange data with many clients via ephemeral ports. To avoid wrong conclusions in the following, we merge such cases into one logical SeS throughout the remainder of this chapter.

5.3.2 Byte- vs. connection-based view of services

Next, we investigate the number of connections, packets, and bytes that are associated with a SeS during the 24 hours of `trace`, see Figure 5.2.

The individual distributions for connections, packets, and bytes shows as

[1] A host is uniquely identified by its IP address.

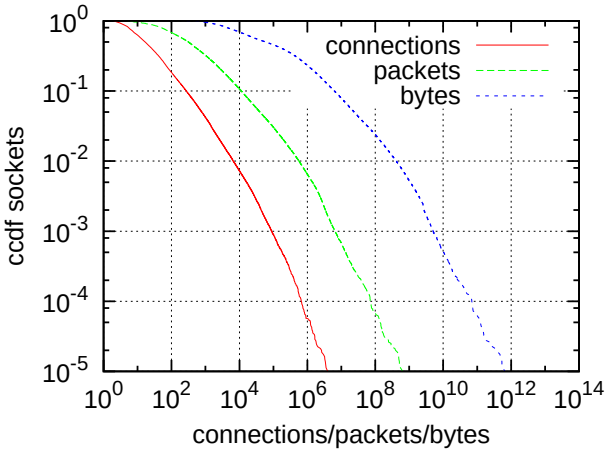


Figure 5.2: *The number of bytes, packets, connections per SeS.*

expected strong correlations. Each SeS attracts during the 24 hours a median of 60 kB, 315 packets, and 17 connections. Surprisingly, 75% of all SeSs experienced less than 1M of traffic during one day.

Spot-checking a large number of these low-traffic sockets reveals that this is not a mistake in our implementation. When accessing such server sockets via `telnet` or via a web browser, we find that many of these SeSs are actually responsive. While Internet traffic may concentrate on a few SeSs with large traffic, there still exists a very large number of network applications that are only rarely accessed by a small set of users, or are not persistently used. We believe that a network-wide view of traffic must include or at least be aware of such low-traffic SeSs. On the other hand, our result that there are few high-traffic SeSs allows network operators to monitor those "important", persistent services with reduced efforts. For example, they can actively ping such services and raise an alert in case of unreachability.

Based on the preceding findings, one can speculate that low-traffic SeSs always correspond to services that very few users access. To check, we rank our detected SeSs by two different criteria: (i) by bytes and (ii) by the number of distinct sockets with which they have established a connection. The scatterplot of Figure 5.3 compares both ranks. SeSs on the black line have the same rank in terms of both criteria.

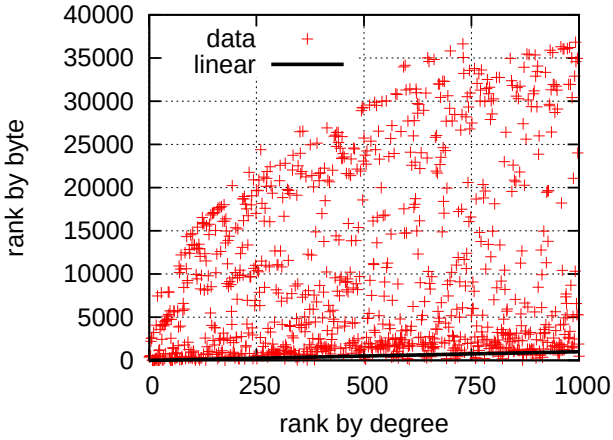


Figure 5.3: *Server sockets: degree rank vs. byte rank.*

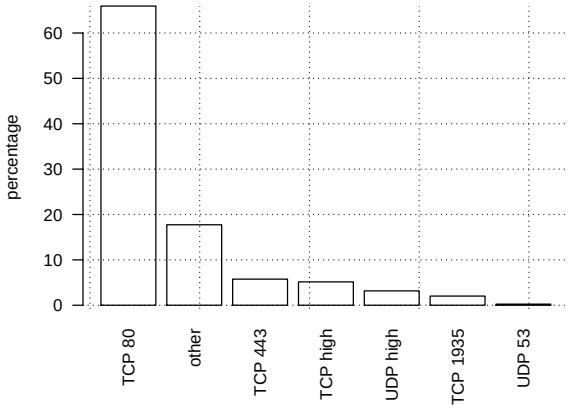
Certainly, there exist SeSs with high traffic volume but only a few "customers": For many SeSs the byte rank is larger than the degree rank, i.e., is significantly below the line. However, there are also many sockets where the degree rank is significantly larger than the byte rank. Despite attracting little traffic, such low-traffic, high-degree SeSs can correspond to critical services that are accessed by many users and that network operators may be interested to monitor. Indeed, we find important applications such as DNS, NTP, etc. that hide behind such SeSs. Therefore, we argue that a server-socket-based perspective can add great value to widely used byte-based traffic views.

5.3.3 Port-based analysis: traffic volume vs. server socket

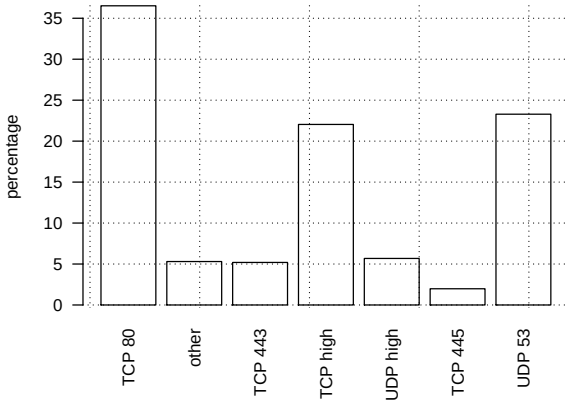
To highlight the differences between a pure byte-based and server-socket based traffic view, we break down observed traffic towards our server sockets by port numbers, see Figure 5.4. We distinguish between some well-known ports[2](HTTP:80, HTTP:443, DNS:53, Flash:1935), traffic on ephemeral ports above 1023 (UDP and TCP not listed), and remaining traffic ("other").

In accordance with recent studies [24, 92], we find that port 80 traffic dominates in terms of traffic volume (more than 50 terabyte for 24 hours,

[2]We relied on a list of known ports taken from [153]



(a) Byte distribution per port.



(b) Server socket distribution per port.

Figure 5.4: *A port-based traffic analysis.*

label	count	percentage
skype	18198	45.5%
bittorrent	8931	22.3%
tcp	8703	21.7%
udp	2613	6.5%
other	1490	3.7%

Table 5.2: *DPI labels for high-port server sockets (considering only server sockets that could be labelled).*

60%). This is due to the popularity of streaming and video applications, and also due to the migration from P2P file sharing applications to web services [24]. Other port numbers with high traffic are Flash and HTTPS. Only around 8% in terms of bytes are associated with network applications that run on ephemeral, high port ranges (UDP high + TCP high).

This picture changes if we break down all our observed server sockets by the transport layer port and protocol used, see Figure 5.4(b). Now, HTTP SeSs make up around 36% (less than 30 million) of all our SeSs. The large number of DNS SeSs is due to the fact that SWITCH acts as registry for .ch and .li domains. Surprisingly, we now find that almost 30% of all SeSs run on ephemeral, high port ranges (TCP: 22%, UDP: 6%). In a nutshell, the port-based application mix strongly differs for the byte-wise and server socket-wise analysis.

5.3.4 High port, end-user deployed services

One can argue that the large number of SeSs on ephemeral, high ports is an artifact due to our implementation, or that such server sockets are not necessarily interesting for network operators. In the following, we refute these arguments by shedding more light onto the specific applications that a SeS provides.

We start by extracting all SeSs that reside in high port ranges (above 1023). Then, we rely on our labelled tuples from the DPI appliance to determine the application behind each SeS. Since our DPI labels are solely available for one customer of the SWITCH network (ETH Zurich), we can only label 17% of all these high-port server sockets. Moreover, we point out that the precision of the application labels varies. Sometimes, the classification can be very generic (e.g., label “TCP”) or even unsuccessful (category“other”).

Table 5.2 summarizes the results.

Surprisingly, we find that more than 45% of high-port SeSs that could be labelled are detected as Skype and another 22.3% as Bittorrent. Apparently, these Skype or Bittorrent SeSs correspond to supernodes. Both applications are not classical network services in the sense that they are reachable over long time periods such as months or years. Yet, tracing such services can be very valuable. After all, Skype is widely used for business reasons such as conference or video calls. Importantly, our agnostic approach is able to detect such services irrespective of the actual application, and therefore can even detect new service types. In a nutshell, based on SeSs we find that end-user deployed network services (Skype, P2P, etc.) are more pronounced than widely thought, although they make up only a small share of the total traffic volume.

5.3.5 Occurrence Frequency

Our approach delivers a list of SeSs that are currently active, or more precisely that are exchanging data with at least one other socket during the studied time interval. So far, we have not analyzed how persistently server sockets are used across time. To analyze this, we rely on `trace` that spans 24 hours but partition the trace into 24 bins of one hour each. We iterate over all our SeSs, and count the number of time bins for which it has exchanged data. Figure 5.5 displays the results.

Slightly more than 30% of all SeSs are observed only during one hour. This is surprising, since it implies that many of our SeSs are quite short-lived. Another possible explanation is that many networks users are inactive (e.g., during night) and some SeSs thus do not receive any "customers". However, high-volume or high-degree services that are mainly interesting for network operators show more stability. Despite times of user inactivity (e.g., nights), around 5% of the SeSs are accessed during more than 20 hours. Importantly, these sockets explain more than 80% of total byte volume and more than 70% of the total connections that we observe. Crosschecking with other days, we obtain similar findings. Apparently, there seems to exist a certain stability or "normal usage" for the SeSs that matter for network operators.

Therefore, we believe that monitoring SeSs paves the way for new network monitoring applications in the area of anomaly, intrusion, and botnet detection [80,122]. For example, in Fast Flux attacks [116], where DNS records are frequently updated, our techniques can be used to discover a sudden rise

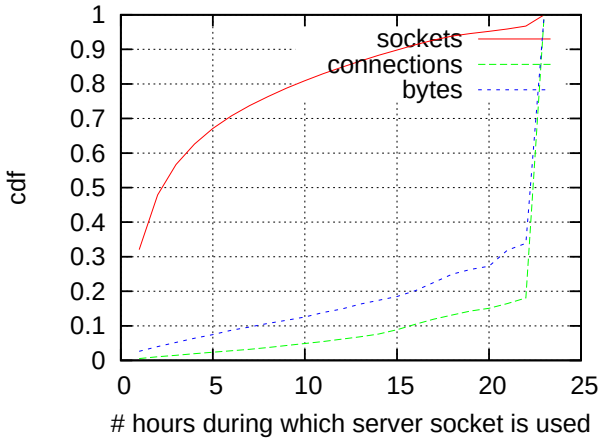


Figure 5.5: *The occurrence frequency of SeS across 24 hours.*

in the number of bytes or connections on a DNS SeS. In addition, these SeSs that are persistently active over time acts as an interesting resource to track connectivity problems. For example, if a stable SeSs is not reachable anymore this is a sign for a connectivity problems. We discuss in Chapter 9 in more detail how this information can be used to track connectivity problems with remote autonomous systems, networks, and hosts.

5.3.6 Coincidence of Network Services

Port-based traffic classification seems to be in a dead end given that more and more services are run over HTTP(S) [99]. This is definitely true if one tries to infer the underlying network application from a single port. Our idea, however, is to consider the proximity of services on the same host or even subnet. To illustrate this, we group our detected SeSs by host, and manually study some typical port patterns that include the HTTP port 80. Table 5.3 reports four example patterns we found.

The Internet mail system relies on multiple protocols such as SMTP, IMAP, POP, webmail, etc. While most of these mail protocols can be identified directly by their port number, it is not obvious how to infer that a certain web server provides also a webmail interface. Yet, if a port 80 server socket

Type	ports
(Web)mail	TCP: 25, 80, 110, 143, 443, 587, 993, 995
VPN over HTTPS	TCP: 80, 443, UDP: 443, 500, 4500, 10000
Streaming	TCP: 80, 1935

Table 5.3: *Typical port patterns for 3 example applications.*

coexists with other mail-related SeSs (such as SMTP on port 25 or POP on port 110), it is likely that the port 80 SeS represents a webmail interface. In a similar manner, the coincidence of SeSs for key management (e.g., Internet Security Association and Key Management Protocol on port 500) next to an HTTPS socket (443) on the same host can suggest that this host offers HTTPS based VPN service. Finally, port 1935 (Flash) next to port 80 points to web pages with embedded video content such as YouTube. In a nutshell, all these examples illustrate that studying the complete set of SeSs and used ports on a given host allows for a more detailed characterization of the role of a host. Most importantly, no deep packet inspection is required for this.

To explore the potential of such service classification techniques further, we intend to rely on the labelled tuples from the DPI appliance, see Section 2. Given a list of detected SeSs, we rely on the information from our DPI appliance to obtain as many SeSs as possible that belong to a certain type (e.g., Skype or Bittorrent supernodes, VPN server, etc.). We identify the hosts (server hosts) on which the extracted SeSs reside, and collect for each such server host the list of all SeSs. Based on the set of all transport-layer ports used on a server host, we study coincidence sets, i.e., transport-layer port numbers that frequently coincide on server hosts of the same type. The problem of finding coincidence sets is a well studied problem in the data mining research community. To this end, we have applied the A-priori frequent data set mining algorithm [14]. To make the data compatible for the coincidence analysis, we associate a label `TCP-X` or `UDP-X` to a server host if the server socket is a TCP or UDP socket, with port number X. We are using port U to indicate a port number is not listed in [153][3]. Since we sometimes find a TCP and UDP socket with the same port number X on a server host, we "aggregate" them to the same item `TWIN-X`. Further we used `SEQ` to indicate that a server host have multiple SeSs on a narrowed port range such as UDP-6600, UDP-6601, and UDP-6602.

[3]This list is maintained by the community and includes as well application that are not officially registered by IANA

Type	port pattern
Skype	80 + 443 + TWIN
Bittorent	TWIN + TCP-SEQ
AOL Instant Messenger	443 + TCP-H

Table 5.4: *Frequent itemset mining – 3 examples patterns.*

Table 5.4 provides three example patterns that we found. First, based on our DPI labels we observe that Skype services frequently reside on server hosts with port 80, port 443, and the TWIN signature. The latter exists if a UDP and a TCP socket run on the same host with the identical high, ephemeral port number (above 1023). Second, Bittorrent SeSs are likely to have such a TWIN signature together with the TCP-SEQ signature, which represents multiple SeSs with consecutive port numbers on the high-port range. Finally, a combination of HTTPS and a high TCP port (TCP-H) frequently points to an instance of the AOL Instant Messenger.

Service classification is not the main topic of this chapter. Therefore, we leave it for future work to perform a more comprehensive study. However, we discuss in the next chapter how this observations can be integrated into a service classification method to label HTTPS SeSs.

5.3.7 Long Term Evolution

We now turn to our study of network applications and their evolution over the last years. For this purpose, we apply our proposed techniques to detect SeSs relying on our 5-day traces from the years 2003 to 2010, see Section 5.2. In addition to aggregate traffic statistics, we present our results on the number of observed SeSs, the amount of traffic per socket, and the port ranges used. In contrast to other research, our study is not restricted to well-known ports from low port numbers, but also takes into account high port numbers that are used by P2P-like applications such as BitTorrent or Skype.

Traffic Volume

In Figure 5.6(a) we briefly revisit traffic volumes observed from 2003 to 2010 in our 5-day traces, displaying the total amount of traffic observed for the major port numbers. Note that we have relied on a list of known ports taken from [153] to assign the underlying applications. A first glance shows that the

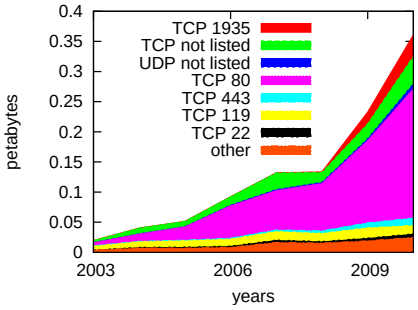
traffic has undergone an increase by a factor of more than 18.3, reaching 0.36 petabytes in 5 days in 2010. The HTTP traffic share has increased steadily from 27% of traffic in 2003 to 49% in 2007, and to more than 59% in 2010. This confirms that port 80 traffic is becoming increasingly dominant nowadays [24,92]. Most likely, this is due to the popularity of streaming and video applications in the Internet, and also to the migration from P2P file sharing applications to web services [24]. Other port numbers with major growth in activity are 1935 (Flash, video streaming), 119 (NNTP, used for file sharing), and 443 (HTTPS). A total of 15% (2003: 17%) in term of bytes cannot be associated to any listed application in [153].

Number of Server Sockets

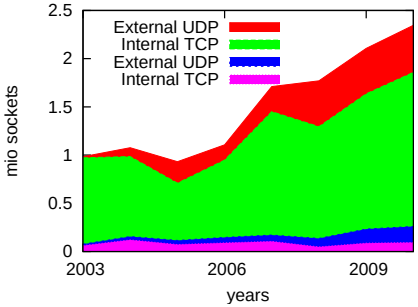
We plot in Figure 5.6(b) the number of detected SeSs between 2003 and 2010. We notice that their total number has more than doubled during the last 8 years, while simultaneously traffic volume has grown by a factor of more than 18, see Figure 5.6(a). This is in accordance with our observation that the number of bytes per socket has slightly increased. Like Labovitz et al. [81], we are convinced that consolidation of content is likely to happen, yet consolidation in terms of deployed network applications does not necessarily occur due to the perpetual increase in SeSs.

Moreover, it is noteworthy that most of the sockets found are outside our studied network (2010: 88.9%, 2003: 92.3%) with a total of 68.2% (2003: 91.2%) being external TCP sockets. Interestingly, the relative increase in the number of external UDP sockets (factor 42.5) has been much stronger than for external TCP sockets (factor 1.8). This trend also holds for internal sockets. The stability of our network across the studied time period of 8 years is illustrated by the fact that the number of internal TCP sockets has not changed significantly.

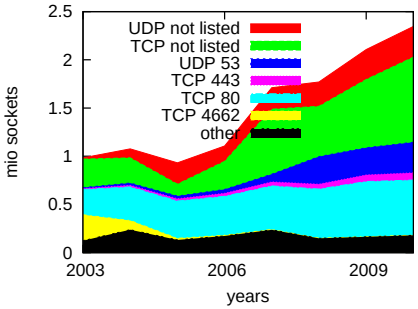
Figure 5.6(c) breaks down all our observed SeSs by the transport layer port and protocol used. The increase in the number of HTTP sockets has been significantly slower than for HTTP traffic. While in 2003, 0.26% of all sockets were HTTP, this number is 0.24% in 2010. Since 2003, we observe an additional number of 307k DNS SeSs, which can point to a higher degree of replication and diversity within DNS. Our proposed techniques also efficiently detect SeSs that are used by P2P-like networks. For 2003 we find 287k (29%) server sockets running on ports 4662 and 18154 that have been widely used by the eMule and Gnutella network, respectively. Interestingly,



(a) traffic volume by port



(b) # sockets by internal/external UDP/TCP



(c) # sockets by ports

Figure 5.6: *The evolution of network services.*

these type of applications almost completely disappeared around 2005. Furthermore, we can observe activity on port 6881, often used by BitTorrent clients, (less than 2%) over all years.

Probably the most surprising finding is the significant percentage of SeSs running on ports not listed in [153], *i.e.* 31.3% in 2003 and 51.1% in 2010. The number of these server sockets has even overtaken HTTP around 2009. We conjecture that this is largely due to the increasing popularity of P2P-like applications used for video streaming or Skype.

Server Socket Location

In order to gain a better understanding of locality of content, we have examined the location of our SeSs by country and AS. To this end, we have relied on country-level information from the MaxMind geolocation database [94] that we have collected since 2003, and we have used BGP routing information to map IP addresses to AS numbers. Further, we used the Policy Block List (PBL) from Spamhaus [134] to get hints whether a host is within an end-user IP address range or not.[4] For anonymity reasons, we can only discuss high-level trends.

We find that a large number of SeSs are located far away from the end users of our studied network. Around 30% of our sockets reside on a different continent than the studied network. Evidently, one reason is that the user base of our ISP is very diverse, including people from countries all around the world. Fortunately, this improves the coverage of our study in terms of SeSs significantly.

Overall, there is a clear trend away from country-wise and AS-wise concentration of content. Table 5.5 show the fractions of external SeSs, that are in the top 10 countries, in the top 10 ASes, and that are contained in the PBL. We observe that nowadays around 65% of the external SeSs are located in the top 10 SeS countries while this number was 81% in 2003. Over the complete time period between 2003 and 2010, we observe a similar trend for the number of SeSs that reside in the top 10 ASes (32% in 2003 while 12% in 2010). The number of SeSs and server hosts covered by the PBL slightly increases between 2003 and 2010, potentially suggesting an increasing number of "services" that reside in end-user dominated IP address ranges.

[4] The PBL lists prefixes of DHCP and dial-up networks ranges.

year	sockets	top 10 countries	top 10 ASes	PBL
2003	907k	81 %	32 %	-
2004	920k	77 %	19 %	-
2005	815k	74 %	16 %	-
2006	958k	72 %	16 %	-
2007	1.5m	69 %	17 %	-
2008	1.6m	68 %	13 %	24 %
2009	1.9m	68 %	13 %	29 %
2010	2.1m	65 %	12 %	32 %

Table 5.5: *Percentage of external sockets within top 10 countries, within top 10 ASes, listed in PBL.*

5.4 Related work

Recently, Internet-wide characterizations of network traffic and application usage have caught public attention [15, 81]. The study by Labovitz et al. [81] has been a milestone towards a better understanding of inter-domain traffic and its changes. Our work nicely complements their work. While Labovitz et al. [81] look at network applications on a coarse level (e.g., aggregates such as P2P or web traffic), we study them at the level of individual SeSs and present the evolution for even the last 8 years. Borgnet et al. [15] relied on limited traffic information from the trans-Pacific backbone. The varying traffic conditions on the observed link probably impair the possibility of drawing long term evolution conclusions. Further, Glatz et al. [49] presents the development of one-way flows. Both works mainly present aggregate statistics. Finally, Estan et al [42] proposed already in 2003 approaches for dynamic and automatic grouping flows into traffic clusters, but did not identify server sockets. From their work we borrow the idea of being agnostic about potentially existing network applications.

5.5 Summary

Relying on rich flow-level data, we show *where* network services are located and *what traffic characteristics* they have. The benefits for a ISP and researchers are manifold. First, analysis on the granularity of SeSs pinpoints the existence of end-user deployed services that are more prevalent than widely

thought. Second, studying the stability of services can identify changes in network usage, possibly related to configuration problems. Finally, our approach paves the way for more sophisticated traffic classification that leverages the spatial coincidence of services.

Chapter 6

Service Classification

Many network operators want to annotate network services with labels (e.g., Bittorrent, Skype, WebMail, etc.). Such information alleviates operators to inspect specific network traffic or to exclude traffic from uninteresting network services from further consideration. Our study about the general characteristics of Server Socket presented in the last chapter reveals that web-based mail servers are often colocated with legacy mail services. In this chapter, we show how such insights can be translated into new approaches for service classification. More precisely, we exploit correlations across flows, protocols, and time. Although we limit our study to the labeling of webmail services such as Outlook Web Access (OWA), Horde, or Gmail, the methods proposed can be extended for other services.

6.1 Introduction

Speculations that web browsers could gradually supplant traditional operating systems as the default platform for user applications [160] go back to the first browser war in the mid 90's. According to recent reports [20] many Internet users prefer to access their e-mail via a web-based (webmail) interface such as Horde [142], Outlook Web Access (OWA) [96], or a webmail platform as provided by GMail, Yahoo!, or Hotmail. Similar trends also hold for many other services, such as video, VoIP, and instant messaging, presently making browsers more than ever before the new operating system.

This trend however is detrimental for the field of Internet traffic classification, which over the past years has seen a plethora of proposals [71]. Evidently, *port-based* approaches are insufficient since many of today's applications use port 80 or HTTPS port 443 [99]. *Signature-based* solutions [25, 58, 67, 99, 132], which generally assume packet-level traces, fail if payload encryption is used, raise privacy or legal issues, and may not scale well enough to obtain a network-wide characterization of traffic. Even with hard to obtain, unencrypted full-packet traces, it is very difficult to identify the applications in use [37]. *Statistics-based* (e.g., [6, 10, 33, 87, 95, 98, 118]) and *host-behavior-based* [61, 67, 68] techniques avoid payload inspection. The former rely mainly on flow-level features such as flow duration, number and size of packets per flow and classify traffic flows. In our extensive experiments, we have found that flow-based features are not sufficient to classify individual HTTPS flows as webmail or not. Finally, host-based approaches (e.g., BLINC [67]) attempt to classify a host directly based on its communication pattern, and are the closest in spirit to our approach. To the best of our knowledge, host-based approaches have not been applied to HTTP(S) traffic classification so far.

As a first step towards deciphering HTTP(S) traffic, in this work, we address the challenging problem of labelling HTTPS-based webmail SeSs from coarse-grained NetFlow data. Given its importance for personal and business use, and its exposure to imminent threats [12, 70], there exists a need for a comprehensive view of the Internet mail system [59, 111, 124], including webmail traffic. Measuring webmail traffic can enhance our understanding of shifting usage trends and mail traffic evolution. Surprisingly, we find that more than 66% of all HTTPS flows observed at the border routers of the SWITCH backbone network [143] are related to webmail. With GMail having recently enabled HTTPS by default (and other platforms expected to follow), this number will only increase in the future.

The most important contribution in this work is the introduction and evaluation of three novel features for identifying HTTPS mail SeSs. Key to our approach is that we leverage *correlations across protocols and time*: (i) The Internet mail system is an interplay of multiple protocols (SMTP, IMAP, POP, webmail). We find that webmail servers tend to reside in the vicinity of legacy SMTP, IMAP, and POP servers, which can be identified easily and reliably [72] using port numbers, thus providing significant hints for detecting webmail servers. (ii) Moreover, clients of a mail server share certain characteristics (e.g., usage patterns) irrespective of the used mail delivery protocol,

i.e., IMAP, POP, or webmail. (iii) Finally, webmail traffic exhibits pronounced periodic patterns due to the use of timers (and more generally AJAX-based technologies) to continuously update the displayed information at the client. This work shows that these features can be harvested solely from coarse-grained NetFlow data to classify webmail traffic.

To get a first impression of the ability of the above features to identify webmail traffic, we train a simple classifier based on a (manually) pre-labeled set of hosts. Although finding the best possible classifier is beyond the scope of this work, our simple classifier already exhibits 93.2% accuracy and 79.2% precision in detecting HTTPS servers within SWITCH, which is remarkable given that we rely solely on NetFlow data. Moreover, our work shows that we can also effectively detect webmail traffic (e.g., GMail) towards mail servers *outside* the SWITCH network, for which only a small sample of their total traffic is visible.

Furthermore, we expect our methodology to stimulate advance in the field of traffic classification in general. For example, for our third feature we distinguish between "machine-generated" and "user-invoked" traffic based on network-level data, which is a very general method that can be used for several different types of detection problems.

The rest of this chapter is structured as follows. Section 6.2 explains our data set we have used. Section 6.3 introduces and discusses features to discriminate mail-related traffic from other web traffic. Then, Section 6.4 describes how we construct a classifier based on the features we identify and presents our first results. Finally, we give an overview of related work in Section 6.5 and conclude in Section 6.6.

6.2 Data Sets and Ground Truth

Throughout this chapter we will rely on flow traces that were collected at the border of the SWITCH network (as discussed in Chapter 2). For our studies we have collected a trace in March 2010 that spans 11 days and contains unsampled flows summarizing all traffic crossing the borders of SWITCH. This results in 50 to 140 million NetFlow records per hour, with approximately 3% being HTTPS. Over the 11 days, we observe more than 1 billion HTTPS records.

Instead of analyzing individual flows, we mainly consider *sessions*. Our flow collectors break up long-lived flows into smaller fragments. Therefore,

we merge flows sharing the same 5-tuple (source address and port, destination address and port, transport protocol identifier) if traffic exchange does not stay silent for more than 900s. Moreover, we group merged flows with the same IP addresses into a session if they are not more than $1,800$s apart in time. These values have been determined empirically, and found to work well in the past.

To train a classifier and to verify the accuracy of our classification in Section 6.4 we need a labelled data set. To this end, we use the data set of labelled HTTPS server presented in 2.3 including different webmail applications and non webmail applications such as VPN, Skype, or general WWW.

We like to stress that 66% of the observed HTTPS flows are related to Outlook Web Access, Horde, or other webmail interfaces (e.g., CommuniGate, SquirrelMail) is surprising and emphasizes the high popularity of webmail. Although the numbers may vary, we believe that our finding also holds for networks other than SWITCH. After all, e-mail is an omnipresent service in today's Internet. While we cannot label 5% of the `top500` hosts, we further differentiate 398 non-mail data sources into WWW content (e.g., e-shops, e-learning platforms), Skype, VPN services, etc.

6.3 Features

We are now ready to discuss the features we propose for discriminating between mail related web traffic and other HTTPS traffic. As mentioned earlier, classifying individual flows relying on flow-level statistics alone (e.g., packets and bytes per flow) does not work well. For example, cross-checking the distribution of number of bytes per flow does not reveal any significant differences between webmail and other HTTPS traffic flows. An important challenge for feature extraction is to overcome the inherent heterogeneity of webmail traffic caused by the high number of different webmail implementations.

The features we present in the following are based on two main observations: first, the network-wide view provided by our flow-based data set allows to leverage correlations across hosts and protocols; second, periodic patterns due to regular timeouts are visible in flow data and provide useful application fingerprints.

To this end, this section sketches three approaches for distinguishing webmail from other HTTPS traffic. The key ideas are as follows: (i) classical mail services (POP, IMAP, SMTP) are frequently in the vicinity of webmail ser-

vices (see Section 6.3.1), (ii) the client base of legacy mail services and the clients of its associated webmail service share common behavior in terms of user activity (Section 6.3.2), and (iii) traffic generated by AJAX-based webmail clients shows a pronounced periodicity that we can leverage to extract webmail traffic (Section 6.3.3). The following Subsections explain these methods in detail, discuss their efficiency, and describe how to obtain features that can be used as input for a classifier. [1]

All three feature categories that we present in the following are broad in the sense that the key ideas are not limited to webmail classification. We believe that our features can be more generally applied towards demultiplexing HTTP(S) traffic into individual applications. For example, inferring periodicity of sessions could translate into a powerful tool for distinguishing between "user-invoked" and "machine-generated" (e.g., AJAX) traffic and for extracting flow-level signatures associated with different applications. Also, studying the mix of well-known services within a subnet may provide hints for the existence of other unknown services in the same subnet. Finally, although we will focus on HTTPS traffic when introducing our features (motivated also by the earlier observations about webmail traffic), we stress that the techniques we describe are applicable to both HTTP and HTTPS flow traces.

6.3.1 Service proximity

The Internet mail system relies on the interplay of multiple services or protocols, including SMTP, IMAP, POP, and webmail. We have observed that if there exists a POP, IMAP, or SMTP server within a certain domain or subnet, there is a high chance to find a webmail server in the same subnet. This is often due to network planning reasons or due to the fact that webmail components are often packaged with SMTP based mail applications (e.g. OWA). Since legacy mail delivery traffic and respective servers can be easily and reliably identified using port-based classification [72], we can use their existence to infer webmail servers in the vicinity.

To verify our assumption of server proximity, we detect all mail servers in the SWITCH network relying on port numbers. Overall, we find 300 SMTP, 140 IMAP/S, and 176 POP/S servers. Moreover, we use our `top500` data set, see Section 6.2. For every HTTPS server we compute the IP address dis-

[1] We stress here that simplicity and interpretability, rather than optimality, have been our guiding principles in translating our observations into (scalar) features, used for our preliminary classification results.

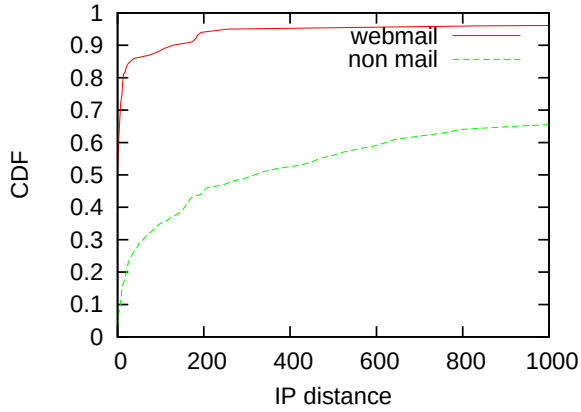


Figure 6.1: *The distance to closest known legacy mail server.*

tance, which is simply the integer difference between two IP addresses towards the closest mail server. Our assumption is that legacy mail servers are closer to webmail than other HTTPS servers. Figure 6.1 shows the observed distance for webmail and non-mail HTTPS servers (e.g., WWW, VPN, or Skype) found among the `top500` servers.

We observe that almost 60% of the webmail servers have a distance very close to the minimum, i.e., smaller than 10 IP addresses, while only 5% of the non-mail servers are in this range. Moreover, more than 90% of the webmail servers have a distance smaller than 200 IP addresses. These numbers indicate that webmail servers are substantially more likely than non-mail servers to be in the neighborhood of a legacy mail server.

6.3.2 Client base behavior

While Section 6.3.1 studies direct correlations between servers (i.e., how close they are), here we are going to analyze the behavior of entire client base of a server. We have found pronounced patterns for the following two properties: (i) the daily and weekly access patterns of HTTPS servers (Section 6.3.2), and (ii) the duration of client connections (Section 6.3.2).

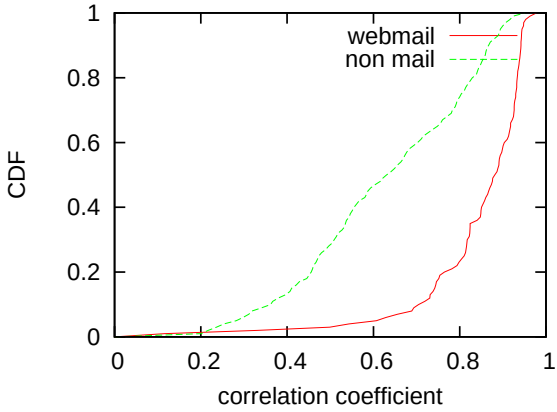


Figure 6.2: *The Correlation between activity profiles of web (mail or non-mail) servers and their closest legacy mail server.*

Daily and weekly profile

It is well-known that Internet traffic as a whole has diurnal and weekly patterns. Different applications may also have their own specific access patterns. For example, people access their e-mail frequently and often in a scheduled way, i.e., first thing in the morning, whereas they access other web applications, such as online banking or Skype, in a substantially different way. More importantly, we expect client activity to be more balanced during the day for servers that have a worldwide user base (e.g., a web page) than for webmail servers with a local user base. For example, everyone can access the website of ETH Zurich, but only a limited group of people can use the webmail platform of ETH Zurich.

To quantify these differences, we use the activity profile of known IMAP and POP servers as a reference point and evaluate how it compares with the activity profile of unknown web servers in their vicinity. Our expectation is that IMAP and POP diurnal and weekly patterns will be more similar to webmail than to non-mail server patterns. In line with the observations of Section 6.3.1, we compare an unknown server to the closest legacy mail server in the same subnet. If there is no such server, we compare to the highest-volume legacy mail server in the same autonomous system.

Specifically, to compare two activity profiles, we partition a given time

interval (in our case one week) into bins of one hour and count for every hour the number of client sessions that access the two servers. This way we derive two vectors of equal length that describe the profile of the servers. To measure statistical dependence between the profiles, we compute the Spearman's rank correlation coefficient [100] of the two vectors. The Spearman's coefficient has two useful properties. First, it is a rank correlation measure and therefore it effectively normalizes the two vectors. Second, it captures (non)-linear correlations between the two profiles, i.e., if the activity intensity of two profiles increases/decreases together. If two profiles are strongly correlated then the coefficient tends to 1. The resulting value is used as input to our classifier (Section 6.4).

To demonstrate the efficiency of this feature, Figure 6.2 plots the distribution of Spearman's correlation coefficient when comparing the activity profiles of all `top500` HTTPS services with the profile of the closest mail server. For more than 90% of the webmail servers the correlation coefficient is higher than 0.6 while this percentage is only about 50% for non-mail servers.

Session duration

A second characteristic of user behavior is how long clients normally use a service. We speculate that webmail users generally take some time to read and answer their e-mails and may even keep the browser window open to wait for incoming messages. Hence, access duration for a webmail service should be higher, on average, than the one for a normal webpage. To capture the access duration of a service we consider sessions as described in Section 6.2. By defining session duration as the time between the start of the first session flow and the start of the last session flow, we ignore potential TCP timeouts. Since we are interested in *typical* user behavior, we use the median across all session duration samples for a given server.

Figure 6.3 displays the distribution of the median duration for our `top500` servers. Again, we find significant differences between the client behavior of a webmail server and other non-mail servers. While the median session duration of a webmail service is shorter than 25s for only some 20% of the webmail servers, almost 90% of non-mail servers experience such short median access times. Although there exist some non-mail servers with long session durations (these are mainly VPN servers), the overall differences in access duration are sufficiently pronounced to use this as another feature for classifying webmail traffic.

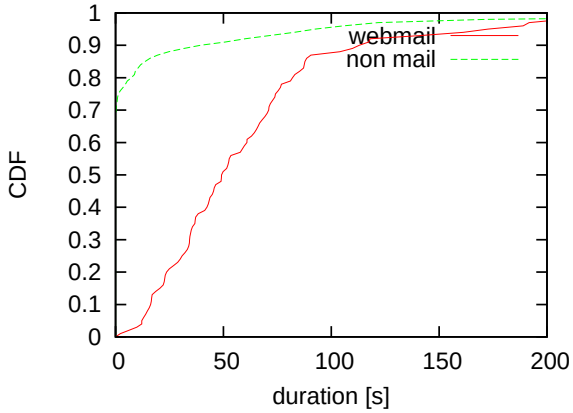


Figure 6.3: *The median of session duration.*

6.3.3 Periodicity

The activity profiles of Section 6.3.2 capture fluctuations in client activity over long time periods such as one week. We now investigate the existence of higher frequency time patterns. The advent of AJAX-based technologies, which heavily rely on asynchronous interactions between clients and servers, has led to an increase in exchanged messages. Many of today's webmail solutions check periodically (e.g., every 5 minutes) for incoming messages and update the browser window if necessary. This is in line with the experience provided by mail programs such as Outlook or Thunderbird. Our idea is to leverage such periodicity that we expect to be visible in our flow-based data in order to classify e-mail related web traffic.

Using `Wireshark` and `Firebug` we analyze the communication triggered by different webmail implementations. We find evidence of distinct periodicity in the sense that even during idle times a synchronization message is sent at regular time intervals. This also results in a new observed flow between client and server at the same time intervals. Nevertheless, capturing this periodicity requires some signal processing.

We observe that webmail sessions are composed of noisy user-invoked traffic, resulting from activities such as sending e-mails or browsing folders, and evident machine-generated periodic traffic. To recover the periodicity buried in the signal, we start by filtering out all short sessions, since they

are not useful for identifying periodic behaviors that repeat infrequently, e.g., every 5 minutes. For a server under study, we keep all sessions and respective flows with a duration higher than 1,800s, which corresponds to six 5-minute intervals. We then split each session into time bins of 1s and count the number of flows that start within a bin. This is repeated for every client of the server under study. Then, we compute the autocorrelation function and average the autocorrelation signals over the entire client base of a particular server, in order to smooth the signal and obtain a server-level behavior.

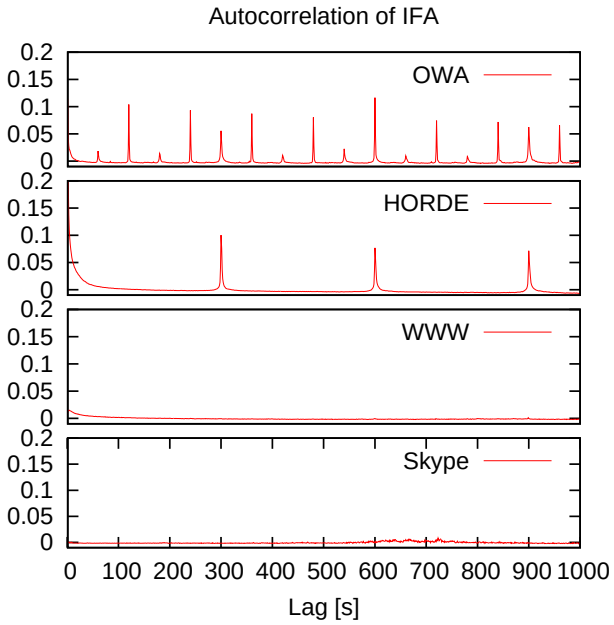


Figure 6.4: *The autocorrelation of flow inter-arrival times.*

The plots in Figure 6.4 display the results for four different types of servers from our `top500` class. Evidently, webmail applications show pronounced peaks at regular intervals, i.e., OWA at 60s and Horde at 300s, while web and Skype traffic does not show this pattern. Note that we observe similar periodic patterns for other webmail applications such as GMail or Yahoo!.

Although it is trivial to assess periodicity visually based on autocorrelation signals, we still need to introduce a simple metric that quantifies period-

icity and can be used by a classifier. To this end, we consider the energies of different frequencies in the autocorrelation function. By cutting off small time lags (e.g., below 200s) and high time lags (e.g., above 400s), we mitigate the impact of noise. Then, we compare the maximum and median signal levels that we observe within this time interval. High ratios are a strong indicator for periodicity, i.e., for the existence of a webmail server.

We believe that the basic idea behind our approach could in general be used to distinguish between user- and machine-initiated traffic. Hence, a spectrum of other applications as well is likely to have a characteristic fingerprint related to flow timing (in fact, our preliminary results already hint to such a direction).

6.4 Evaluation

The results of the preceding Section 6.3 are promising in the sense that it is apparently possible to discriminate between e-mail related web traffic and other HTTPS traffic leveraging correlations across protocols and time. In this section we explain how to build a classifier using our proposed features and present first results. Our main concern is not maximum accuracy. Rather we seek to demonstrate the general feasibility and the potential of our approach in a proof of concept study.

The general approach is as follows: We use the Support Vector Machine (SVM) library provided by [23]. For classification we rely on the four features presented in the preceding Section 6.3, namely service proximity, activity profiles, session duration, and periodicity. These features are used to discriminate between the two classes mail and non-mail. To estimate classification accuracy, we rely on 5-fold cross-validation: we partition the set of hosts into five complementary subsets of equal size. Four subsets are used as training data, the remaining subset serves as validation data. We repeat this process five times such that each of the five subsets is used exactly once as validation data.

Based on the `top500` data set, see Table 2.2, we classify webmail hosts that are inside the SWITCH network.

As shown in Table 6.1, our classification realizes a remarkably high mean accuracy of 93.2% (with standard deviation 3.0) across the five runs of our 5-fold cross-validation. In addition, the precision for the mail class is also reasonably good at 79.2% relying solely on flow data and can be further im-

	Number of true HTTPS	
	mail servers	non mail servers
classified as mail	61	16
classified as non mail	16	382
accuracy (mean)	93.2% ($\pm$3)	

Table 6.1: *Classification of internal HTTPS servers.*

proved by optimizing the classifier. By manually scrutinizing the results of the classification, we find our classifier can effectively discriminate between webmail and Skype nodes mainly due to the service proximity feature of Section 6.3.1 and also between webmail and authentication services mainly due to the session duration feature of Section 6.3.2. However, it is challenging to distinguish between VPN and webmail servers, which is the main reason for false negatives when classifying webmail hosts.

Popular webmail platforms such as GMail, Yahoo!, or Hotmail do not maintain servers within the SWITCH network and, therefore, have been ignored in our discussion and analysis so far. Finally, we briefly illustrate that applying our approach to identify arbitrary webmail applications is equally feasible, irrespective of whether hosts are internal or external.

To this end, we extract additional data from our trace collected in March 2010. Following the methodology outlined in Section 6.2, we select 500 popular *external* HTTPS hosts/sockets. Manually labeling this data set reveals 32 GMail servers[2]. For 452 additional HTTPS servers we established that they provide other non-mail services such as online banking, update, Facebook login servers, etc.

We now merge the original `top500` data set and the labeled external hosts creating a new data set with 1000 HTTPS servers. Based on our proposed features we build a classifier and again apply 5-fold cross-validation.

Table 6.2 shows that we are able to classify internal and external HTTPS servers with overall accuracy of $94.8\% \pm 3$ and precision of 75.8% for the mail class. Although these results are preliminary, it appears possible to detect mail servers outside the SWITCH network using the same features, even though only a small sample of the total traffic of these servers is visible in our network. Moreover, our case study on GMail suggests that the techniques

[2]Recall that GMail recently switched to HTTPS.

	Number of true HTTPS	
	mail servers	non mail servers
classified as mail	94	30
classified as non mail	19	820
accuracy (mean)	94.8% ($\pm$3)	

Table 6.2: *Classification of internal and external HTTPS servers.*

proposed in this chapter are not limited to specific applications (e.g., Outlook Web Access, Horde).

6.5 Related work

Though extensive research has focused on traffic classification[3], only a comparably tiny portion has studied methods for dissecting WWW traffic. Schneider et al. [130] extracted traffic to four popular Ajax-based sites from full packet traces and characterized differences between Ajax and HTTP traffic. Li et al. used manually constructed payload signatures to classify HTTP traffic into 14 applications and discussed trends from the early analysis [86] of two datasets collected three years apart. Compared to previous studies, our work focuses on extracting mail traffic from WWW traces and operates on *flows*, solving a substantially more challenging problem than previous HTTP classification studies, which used full packet traces (e.g., [37]).

Some research has recently focused on the traffic characteristics of mail. Notably, Ramachandran et al. [111] characterized a number of network properties of spammers and emphasized that spammers can alter the content of spam to evade filters, whereas they cannot easily change their network-level footprint. This work suggested the plausibility of network-based spam filtering and was followed by the SNARE system [59], which uses a small number of simple traffic features to block spammers with reasonable accuracy.

Finally, compared to host-based classification, the novelty of our method can be mostly traced to the addition of the following two "dimensions": (i) we introduce a timing dimension, as this contains valuable information about user- and machine- behaviors; (ii) while earlier host-based approaches [67] attempt to identify a *common* communication pattern (graph) shared by all

[3]For a survey of traffic classification literature refer to [71].

hosts of this application, we try to correlate the unknown host behavior to *known* hosts running a different, but *related* application; as a result, hosts with different patterns could be validly identified under one class.

6.6 Conclusion

In this chapter, we have presented a number of flow-level techniques that can be used to separate webmail traffic from other HTTPS traffic. The novelty of our approach goes into two main directions: (i) we leverage correlations across (related) protocols (e.g., IMAP, POP, SMTP, and webmail) and among hosts sharing a similar client base, and (ii) we identify and exploit timing characteristics of webmail applications. Based on these, we have introduced novel features, investigated their efficiency on a large flow data set, and used them to produce preliminary classification results on internal and external HTTPS servers. This is the first work to show that it is possible to identify HTTPS webmail applications solely based on flow-level data with approximately 93.2% accuracy and 79.2% precision.

While the main focus of this chapter has been the presented novel features, in future work, we intend to optimize our classifier and if needed to further extend our set of features (along the two directions discussed) to improve the precision of our classification. In the same direction, we would also like to test the effect of sampling on our techniques, as we believe that most of the features presented are likely immune to sampling. Furthermore, we believe that our features are of broader interest for traffic classification, and could be used, for example, to facilitate demultiplexing of HTTP(S) traffic in general. Finally, having a reliable method to extract webmail traffic, we intend to perform an extensive characterization of the long-term evolution of mail-related traffic including usage trends and traffic mix between classical delivery and web-mail protocols.

Chapter 7

Hybrid Network Service Classification

In the preceding chapter, we leveraged correlations across flows, protocols, and time to introduce novel techniques for classifying HTTPS based network services. These techniques have already been proven to work for large-scale networks. Now in this chapter, we discuss how to achieve service classification with reasonable accuracy and coverage, yet limited effort. Our idea is based on the observation that the service label for a Server Socket (SeS) is stable across multiple flows and different users that access the SeS. Exploiting this fact we propose a new service classification architecture called PARS PRO TOTO [1] that reduces the overall collection and processing efforts without sacrificing more level-of-detail than necessary.

7.1 Introduction

It is a widely accepted saying that "You can't manage what you don't measure". Therefore network professionals (operators, engineers, application designers) need to understand how networks are used to do traffic engineering, to dimension their networks, to troubleshoot problems, to design and develop the next-generation networks or products, etc. But the saying "There ain't no such thing as a free lunch" applies for measuring large-scale networks that

[1] Pars pro toto is Latin for "a part (taken) for the whole"

include thousands to millions of hosts, too. In fact, analyzing large-scale networks at the appropriate level-of-detail to annotate measurement data with labels such as Skype, Akamai, or Bittorrent requires significant collection and processing efforts. For example, collecting fine-granular measurement data (e.g., packet traces) at a large set of vantage points distributed across a complete network often requires a costly dedicated infrastructure. Furthermore, the processing of the voluminous data to apply state-of-the-art traffic classification algorithms is challenging due to today's IO bandwidth limitation and memory access times. Additionally, it is predicted by various studies [28, 73] that traffic volume will increase exponentially withing the next years. Hence, it is likely that traffic labeling methods that still work today will be too costly in the future.

This raises the question how to reduce the overall collection and processing efforts without sacrificing more level-of-detail than necessary. At this point, we like to stress that the goal of this work is not to introduce a new service classification method or to improve the service classification accuracy of specific network service types. In contrary, we analyze how much level-of-detail the service classification needs to be sacrificed to significantly reduce the overall collection and processing efforts. Yet, to the best of our knowledge, only limited work is done in analyzing this trade-off.

To establish this trade-off, we propose to exploit the stability of network services. In more detail, we observed that network services serve users from different parts of the network (address space stability) over longer time intervals (time stability) with the same service. For example, a host running Skype will be contacted by many peers over the time on a specific SeS. If we identify a certain communication toward the SeS as Skype (service classification), we can use this knowledge (service label) to label further communications toward the same SeS that occur during the same time interval as Skype. In addition, since all peers are served by the same Skype process, we can learn this service label from any peer connecting to this host.

In short, the stability allows us to reuse the knowledge learned from a subset of traffic to label the overall traffic. This strategy helps to significantly decrease the overall processing and collections efforts. However, we have to accept that to determine the process behind each SeS it is required to analyze at least one flow towards each SeS in detail. Depending on the measurement setup, this cannot always be achieved causing some traffic to be labeled as unknown, decreasing the level-of-detail available for the operator.

To do a first assessment of the potentials of such an approach we introduce

and evaluate in this work a hybrid measurement architectures called PARS PRO TOTO that is based on two different processing pipes as illustrated in Figure 7.1. First, there is a coarse network sensor which covers the complete traffic of the network or a large share of it, referred to as COARSEPIPE. For example this could be achieved by collecting flow-level measurement data using the router infrastructure of the network. Second, there are sensors that see only a limited part of the overall network traffic, referred to as FINEPIPE. This allows to collect more fine-grained measurement data such as packet or unsampled flows traces. This limited fine-grained data set is further processed to extract service labels.

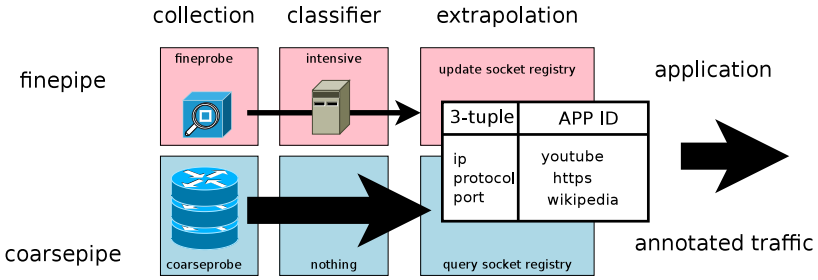


Figure 7.1: *The overview of the* PARS PRO TOTO *approach.*

Our PARS PRO TOTO strategy then correlates both data streams at the level of SeS. In more detail, as soon as a SeS is labeled with the help of FINEPIPE, then this service label is "extrapolate" to the measurement data collected by COARSEPIPE.

For the evaluation of the PARS PRO TOTO approach we rely on the measurement setup introduced in Chapter 2. We instrumented as FINEPIPE the DPI appliance that is attached to the uplink of the ETH Zurich (≈25,000 users). In addition, we use flow-level data that is collected at the border of SWITCH (≈250,000 users), the respective national research and education network (NREN) as COARSEPIPE. Our findings reveal that this extrapolation approach can enhance more than 60% of the NREN's border traffic with details from FINEPIPE, although FINEPIPE sees only 10% of the overall packets. The extrapolation capabilities are surprisingly high even if the covered network size of FINEPIPE is limited, or if FINEPIPE information stems from time intervals hours or days before the COARSEPIPE data was collected.

The rest of this chapter is structured as follows. §7.2 explains our method-

ology and our data. §7.3 is a central section since it investigates the extrapolation capabilities of our PARS PRO TOTO approach. §7.4 briefly discusses applications of our approach. Finally, §7.5 compares against related work, before we summarize our work in §7.6.

7.2 Methodology

In Section 7.2.1 we describe the data sources used for our study. Then, Section 7.2.2 summarizes the data processing needed to shed light into the extrapolation capabilities of our approach.

7.2.1 Measurements Setup

In this work, we rely on the general measurement setup discussed in Chapter 2. In more detail, we rely on flow measurements collected on the SWITCH border routers (see Section 2.1) as source for COARSEPIPE. Further, we use a commercial DPI appliance (see Section 2.2) to obtain more fine-grained information measurement data and to extract traffic labels processed in FINEPIPE.

Throughout the remainder of this chapter, we work with traces spanning five work days from July 4, 2011 (Monday) until July 8, 2011 (Friday). While one day of flow-level data contains some 3.4 billion flows, a respective data set from the DPI appliance includes more than 660 million application tuples, amounting to 55 GB of binary data. Without loss of generality, we will restrict our analysis on TCP and UDP traffic (92.3 % of the traffic). Table 2.1 in Chapter 2 provides an overview of the top 20 application types our DPI appliance detects.[2]

7.2.2 Processing

Based on the above traffic subset, our methodology for extrapolation can be summarized as follows: First, we process the DPI trace and extract SeS, i.e., the side of a connection that is marked by the appliance with a server flag. We keep track of all identified server sockets in a *socket registry*. Its hash-like implementation uses the 3-tuple IP address, port number, and protocol as key. Stored values include the information that we want to scale up, in our case the

[2] It is not the goal of this work to judge the accuracy of the achieved traffic classification.

application identifier of the socket. Unfortunately, our DPI appliance classifies parts of the traffic with imprecise labels such as TCP or UDP.[3] As our goal is not accurate traffic classification and since the socket registry monitors application types of server sockets across longer time periods, we decide to ignore all server sockets that exclusively have TCP or UDP labels in the following.

Afterwards, we process the flow trace, which we match against the list of known server sockets. From our SWITCH flow-level traces, we extract both sockets, i.e., the two endpoints from a flow, and check if they are already contained in the socket registry. If yes, we can use the stored information (e.g., application label) to extrapolate for example byte/packet counter statistics to the overall network, or to filter based on the type of application.

Regarding the performance and scalability of the *pars pro toto* approach, our first experiences clearly suggest that real-time monitoring is feasible. In our test setup, we need 9 hours to process 24 hours of data on a Dual Core AMD Opteron 275 using only one process and less than 2 GB memory.

7.3 Findings

Using the presented methodology, we assess in this section the potential of the proposed PARS PRO TOTO approach and study different parameters that will impact this potential.

In more detail we first revise the stability of the collected application labels, a fundamental requirement for the success of our approach. Then, we provide first insights about the fraction of traffic observed by COARSEPIPE that can be "extrapolated" relying on knowledge from FINEPIPE. Afterwards, we discuss in Section 7.3.3 how this potential depends on the update rate. Furthermore, in Section 7.3.4 we investigate how the extrapolation potential could change if we apply our approach on a different network having a different application mix.

7.3.1 Stability of Labels

The type of network application on a certain socket should remain the same for a reasonable amount of time (e.g., minutes and hours). Otherwise, we

[3] We reckon that this is mainly due to malformed traffic, e.g., empty packets in network attacks.

cannot conclude that traffic characteristics observed in FINEPIPE at time t must be similar for traffic towards the same sockets in COARSEPIPE at time $t+x$, where x is in the magnitude of minutes or a few hours.

It is widely agreed that a specific port only provides one functionality at a given time, e.g., being a HTTP or FTP server. In order to check the stability of a network application on a specific socket, we perform the following analysis: we compare the observed application labels (see Section 7.2.1) for given sockets by extracting these labels separately for individual days. For each socket, we take the set of application labels seen on Monday as our reference point.

We find that between $92-95\%$ of application labels (same src/dst IP, same src/dst port, same application ID) show up again in the remaining four days. In general, we learn *only* for less than 0.5% of the respective server sockets a different application label between Tuesday and Friday. This underlines that most server sockets generally do not change the type of service they offer and stay stable even over very long observation intervals.

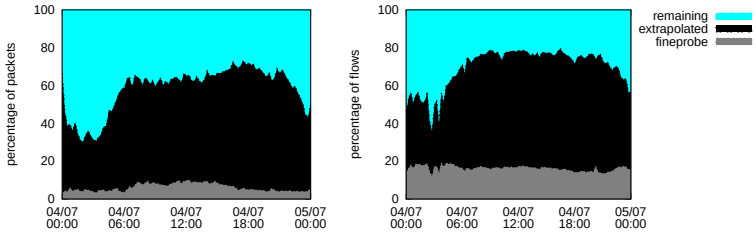
7.3.2 Coverage

There should be enough overlap between the (server) sockets in FINEPIPE and COARSEPIPE. In our example: if most flows from SWITCH connect to hosts other than the ones for which we have already an entry in the socket registry (based on ETHZ information), then the utility of extrapolating traffic characteristics to the overall network will remain restricted.

Now, we evaluate this core aspect of our approach, we define *extrapolation potential* (EP) as the fraction of traffic in terms of bytes or packets for which it is possible to extrapolate based on the content of the socket registry.

The stacked curves of Figure 7.2 present the results for July 4, 2011. For every 15 minute time interval, we show the fractions of the traffic that (i) is labelled traffic from ETHZ ("finepipe"), (ii) that can be extrapolated relying on the detected server socket ("extrapolated"), and (iii) finally the remaining traffic ("remaining").

We find that the observed extrapolation potentials for bytes and packets are highly correlated, and decide to present only plots for bytes and flows, see Figure 7.2. In both cases, the extrapolation potential undergoes strong daily fluctuations, being significantly lower over night. We believe that this is due to the high relative importance of "background noise" (e.g., scanning) at night, when overall traffic volume is lower. Yet, throughout daytime, the cate-



(a) By bytes (b) By flows

Figure 7.2: *The extrapolation potential.*

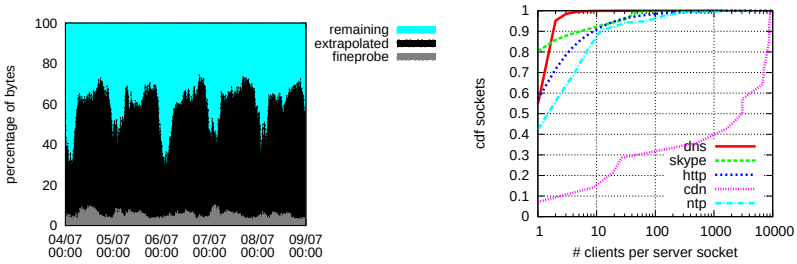
gory "extrapolated" makes up a significant part, averaging slightly above 60% for bytes and packets (70% for flows). This is surprising given that labeled ETHZ traffic makes up only around 10% of the SWITCH border traffic and given the high diversity of the SWITCH network with a trilingual user community (German, French, and Italian). With respect to the latter, we expect even higher extrapolation potentials in more homogeneous environments.

In addition, we assessed the extrapolation potentials separately for the individual days between July 4 and July. The measurement reveals that the fraction of "extrapolated" SWITCH border traffic (in terms of bytes) is more or less the same for the five work days: Monday 62%, Tuesday 63%, Wednesday 63%, Thursday 67%, Friday 64%. These findings suggest that PARS PRO TOTO approach performs equally well and is not time sensitive.

Finally, we point out that in practice, one could collect labelled traces at any site or even at multiple sites trying to achieve a good tradeoff between optimal extrapolation potential and deployment efforts. Future technologies such as OpenFlow [104] could further help to dynamically mirror or redirect parts of the traffic to FINEPIPE in order to improve the coverage for certain socket types on demand (e.g., to further improve the coverage for p2p system like Skype).

7.3.3 Degradation over Time

The collection and processing efforts can be further reduced by only periodically using FINEPIPE to update the data of the socket registry. However, this raises the question how the extrapolation potential degrades if we stop collecting more data from FINEPIPE. To evaluate this, we assume that the socket



(a) To time interval covered by FINEPIPE

(b) To spatial coverage of FINEPIPE

Figure 7.3: *The sensitivity of the extrapolation potential.*

registry only contains information about server sockets visible in FINEPIPE during the first day, i.e., July 4. For the remaining four days, extrapolation continues but is based on "out-dated" information from Monday. Similar to Figure 7.2(a), we display in Figure 7.3(a) the extrapolation potential in terms of byte rates. Interestingly, there is only little degradation for Tuesday until Friday, and the extrapolation potential in terms of bytes stays around 60%. This is due the the fact that the majority of the popular sockets are still alive on Friday. This result suggests that the collection and processing efforts can be indeed reduced by updating the socket registry only periodically.

7.3.4 Application Sensitivity

Certainly, it is possible to maximize the extrapolation potential by choosing appropriate sites and locations for the measurement of FINEPIPE. For our study, we cannot add vantages points or change the location where FINEPIPE is measured. Yet, we can safely assume that choosing vantage points is not that critical if the important, popular network services (server sockets) are widely accessed from many different hosts and subnets.

To verify this assumption, we count how many distinct IP addresses inside ETHZ have established communication with server sockets outside the ETHZ network during the 24 hours of July 4. Figure 7.3(b) displays the cumulative distribution if we break down by DNS, Skype, HTTP, Akamai CDN, or NTP server sockets. As we can see, the distributions for these network applications differ significantly. While more than 85% of Akamai CDN server sockets (HTTP: 15%) have more than five clients, this number is considerably

smaller for DNS or Skype. The reason is that Skype adopts a P2P-like communication mechanism, leading to many Skype sockets having only a few clients, probably limited by the number of friends inside the contact list. Yet, there also exist 700 Skype server sockets with more than 100 clients. These are likely to be supernodes and can be relevant for extrapolation. Regarding DNS, our FINEPIPE sees very few external DNS server sockets because DNS requests inside ETH are almost exclusively handled by a few ETHZ-internal DNS nameservers that communicate with the outside world. Therefore, care has to be taken if one wants to collect a FINEPIPE that captures DNS traffic. In general, if FINEPIPE does not cover a certain class of traffic, it is impossible to learn labels for it that can be used later for extrapolation.

Apparently, those server sockets that contribute most to traffic volume (e.g., CDN content servers, popular web servers) are accessed by a lot of clients. This insight is crucial, indicating that having a FINEPIPE with limited spatial coverage (in our case around 10%) can still result in good extrapolation.

7.4 Applications

After studying the extrapolation capabilities, we now briefly discuss first applications that could benefit from a hybrid, *pars pro toto* measurement approach. In fact, our approach can reveal the composition of network traffic for *large-scale* networks without network-wide deployment of fine-granular network sensors. We point out that it is not our goal to propose accurate traffic classification techniques. In the following, we therefore assume the correctness of traffic labels from FINEPIPE, and perform only some minor processing of the labels. For example, we group all traffic from Akamai content servers, which can be any type of HTTP-based web content, into “Akamai CDN”.

The Tables 7.1, 7.2, and 7.3 briefly summarize traffic statistics by bytes packets, and flows that we can predict for the complete traffic that SWITCH exchanges with external networks.

Consistent with previous findings [92], we find that HTTP traffic dominates by a significant margin (46.73% in terms of bytes). Here, we point out that we do not include applications for audio and video transmission such as Shoutcast in the HTTP category. Regarding traffic volume, content providers (Akamai[4] 28.20%, Google 2.6%), streaming (Shoutcast 8.06%),

[4]The high amount of Akamai traffic is due to an Akamai edge server inside the SWITCH

top	application	bytes (%)
1	http	46.73
2	akamai cdn	28.20
3	shoutcast	8.06
4	https	3.88
5	google cdn	2.60
6	rtmp	1.40
7	ssh	1.18
8	ipsec	1.11
9	megaupload	0.71
10	bittorrent	0.69

Table 7.1: *The predicted top 10 applications by bytes for the SWITCH network .*

or Flash videos (RMTP 1.40%) significantly contribute. In general, there is strong correlation in the statistics for bytes and packets, and even with the flow ranking. Yet, we can see that Skype makes up a large fraction of all flows, but is not that important packet- or byte-wise. The high share of DNS flows can be explained since SWITCH serves the `.ch` and the `.li` top-level domain.

It is true that the SWITCH network is of academic nature, and thus it is not clear how well it could represent the traffic mix of other, commercial networks. Yet, we note that the accuracy of the computed traffic mix is not of key interest. Rather, our focus is on illustrating one important application of our methodology, namely determining the network-wide composition of traffic.

7.5 Related work

We acknowledge that hybrid traffic classification techniques have already been proposed in the past [108, 157]. Yet, the focus has not been on the scaling up of measurement results. While literature has widely discussed self-similarity of network traffic at different time scales [34], there is significantly less research on spatial similarity. The work from Pietrzyk et al. [107] relies on a DPI appliance to train a statistical classifier, and then checks the accuracy

network that is used by a commercial ISP to download content

top	application	packets (%)
1	http	44.54
2	akamai cdn	24.79
3	shoutcast	7.81
4	https	5.21
5	google cdn	2.64
6	rtmp	1.64
7	skype	1.55
8	dns	1.42
9	ipsec	1.28
10	ssh	1.23

Table 7.2: *The predicted top 10 applications by packets for the SWITCH network.*

top	application	flows (%)
1	dns	32.80
2	http	23.20
3	akamai cdn	11.08
4	skype	9.95
5	ntp	8.29
6	https	5.47
7	google cdn	1.71
8	facebook	1.21
9	bittorrent	0.91
10	google	0.67

Table 7.3: *The predicted top 10 applications by flows for the SWITCH network.*

if the same classifier is applied to traffic recorded at a different site of the network. Their results are based on a significantly smaller user base (some 2000 users per trace), and suggest that it is difficult to apply a statistical classifier for a site for which it has not been trained.

7.6 Summary

We proposed a *hybrid* approach to monitor large-scale networks at a level-of-detail that can provide meaningful insights at reasonable efforts. One workflow (FINEPIPE) monitors network traffic at high resolution (e.g., packet-level), but only sees a limited network part. In addition, the coarse workflow (COARSEPIPE) collects traffic statistics for the complete network, yet at a coarser granularity (e.g., flow-level) without mining the data.

Our *pars pro toto* strategy then correlates both data sources at the level of sockets, and complements information from COARSEPIPE with details from FINEPIPE. In our case study based on the SWITCH and ETHZ network, we find that signficantly more than 60% of the NREN's border traffic can be enhanced with traffic labels, although FINEPIPE sees only 10% of the packets.

Part III

Troubleshooting Services in the Wild

Chapter 8

Tracking Mail Service

After describing our annotation techniques, we turn to the first troubleshooting application. We focus on the e-mail service due to its importance as communication medium. We discuss how flow-based information collected at the network core can be instrumented to counter the increasing sophistication of spammers and to support mail administrators in troubleshooting their mail services. In more detail, we show how the local intelligence of mail servers can be gathered and correlated *passively*, scalably, and with low-processing cost at the ISP-level providing valuable network-wide information. First, we use a large network flow trace from a major national ISP to demonstrate that the pre-filtering decisions and thus spammer-related knowledge of individual mail servers can be easily and accurately tracked and combined at the flow-level. Then, we argue that such aggregated knowledge does not only allow ISPs to monitor remotely what their "own" servers are doing, but also paves the way for new methods for fighting spam.

8.1 Introduction

According to IronPort's 2008 Security Trend Report [63], as much as 90% of inbound mail is spam today. Moreover, spam is no longer simply an irritant but becomes increasingly dangerous. 83% of spam contains a URL. Thus, phishing sites and trojan infections of office and home systems alike are just one click away. The rapid increase of spam traffic over the last years poses significant processing, storage, and scalability challenges for end-host sys-

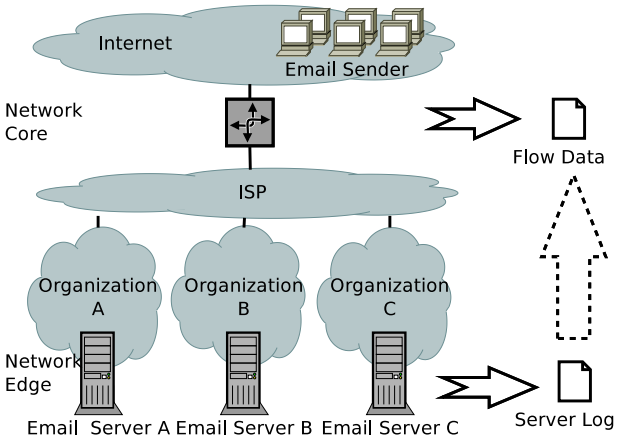


Figure 8.1: *The ISP view of the network.*

tems, creating a need to at least perform some fast "pre-filtering" on the email server level. To do this, email servers evaluate information received at various steps of the SMTP session using local (e.g., user database, greylisting [60]) and global knowledge (e.g., blacklists [133,135] or SPF [158]) to identify and reject malicious messages, without the need to look at the content.

Nevertheless, traditional pre-filtering methods like blacklists are starting to lose their edge in the battle. Spammers can easily manipulate an IP block for a short time to do enough damage before they can be reported in a blacklist [39, 110]. To amend this, new filtering approaches focusing on general network-level characteristics of spammers are developed [11, 31, 56, 112], which are more difficult for a spammer to manipulate. An example of such characteristics are geodesic distance between sender and recipient [137], round trip time [11] or Mail Transfer Agent (MTA) link graph properties [36, 50]. These methods have been shown to successfully identify additional malicious traffic that slips under the radar of traditional pre-filtering. Yet, they require different amounts of information and processing, ranging from simply peeking into a few entries of the packet header to less lightweight, more intrusive approaches.

Our work is in the same spirit, in that we are also interested in the network-level characteristics of spammers. However, we look at the problem from a somewhat different perspective. Specifically, we look at the problem from an

AS or ISP point of view comprising a network with a large number of email servers. We assume that a number of servers in this network (if not all) already perform *some* level of pre-filtering, e.g., dropping a session to an unknown recipient, using a blacklist, or even using sophisticated network characteristics based mechanisms like the one proposed in [137]. This essentially implies that (a) each server is not performing equally "well" in identifying and blocking spammers, and (b) each server has a limited, *local* view or opinion about which senders are suspicious or malicious. In this context, we're interested in answering the following question: *can one use a* 100% *passive, minimally intrusive, and scalable network-level method to (a) infer and monitor the pre-filtering performance and/or policy of individual servers, and (b) collect and combine local server knowledge in order to re-use it to improve server performance?*

Although one could potentially use individual server logs to gain the needed pre-filtering information, in order to collect network-wide spam statistics an ISP would have to gather the logs of all mail servers in the network. As these servers are usually located in many different organizational domains, this is a tedious process that is hindered by privacy concerns of server operators. Instead, *we demonstrate that the pre-filtering decisions of individual servers can be passively and accurately inferred in the network using flow size information captured in the network core* as illustrated in Fig. 8.1. Having validated this methodology, we then use it to analyze the incoming SMTP traffic of a major national ISP network with 320 internal email servers. We found that internal servers perform very differently. Some servers accept up to 90% of all incoming SMTP flows, while many accept only $10-20\%$. We look further into the causes of these discrepancies, and after ruling out various "benign" causes, we conclude that many servers in the network seem to be misconfigured or simply under-performing. Based on this, we investigate how and to what extent the *collective* knowledge of well-performing servers could be used to improve the pre-filtering performance of everyone.

Summarizing, our method avoids the cumbersome process of log gathering and correlation. It also requires minimal processing and session information, implying that this method is scalable enough to keep up with the high amount of information constantly gathered at the network core. Finally, it is complementary to recently proposed, sophisticated spam detection mechanisms based on network characteristics, in that the whole system could benefit from such increased capabilities deployed a given server or subset of them.

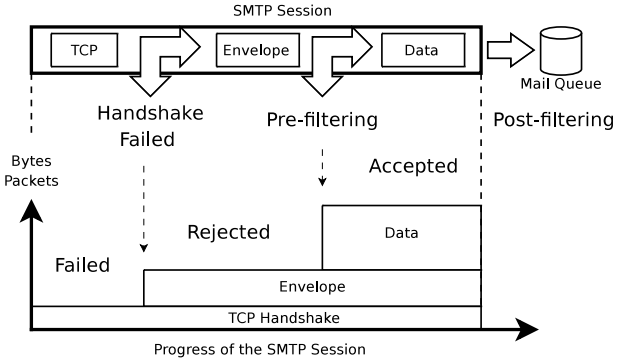


Figure 8.2: *The three phases of email reception.*

8.2 Preliminaries

The email reception process on a server consists of three phases as depicted in Fig. 8.2, TCP handshake, SMTP email envelope exchange, and email data exchange. Pre-filtering is employed in the second phase: in order to identify and quickly reject malicious traffic based on the message envelope only a server may use "global" knowledge (e.g., sender listed in a blacklist), local knowledge (e.g., attempt to reach unknown recipients), or policy-based decisions (e.g., greylisting).

We analyzed the log of a university mail server serving around 2400 user accounts and receiving on average 2800 SMTP flows per hour to look into such pre-filtering performance in more detail. We found that as much as 78.3% of the sessions were rejected in the pre-filtering phase. 45% of the rejects were based on local information (e.g., user database or greylisting) and only 37.5% were due to blacklists. This illustrates the importance of local mail server knowledge for spam detection.

Based on the server's decisions, we classify observed SMTP sessions as either *failed*, *rejected* or *accepted*. Our key observation is that, whenever a sender manages to get to the next phase, the overall transferred information is significantly increased. For example, if a sender is accepted and allowed to send email content, he is able to transmit much more data than a sender already rejected in phase two. As a consequence, we conjecture that flow properties reflecting the size or length of SMTP sessions, such as the flow

size or packet count, should be an accurate indicator for the phase in which an SMTP session was closed.

We validate this assumption in Section 8.3. For this purpose, we have used three weeks of unsampled NetFlow data from January, February and September 2008 (referred to as week 1, 2, 3), captured at the mesurement setup discussed in Section 2. The identification of SMTP traffic is based on TCP destination port 25[1]. Based on the SMTP traffic, a list of active internal email servers was generated and verified by active probing. We detected 320 internal servers, receiving up to 2 million SMTP flows per hour.

8.3 SMTP Flow Characteristics

In this section, we demonstrate how the effect of pre-filtering on flow characteristics can be used to track the servers' decisions for each SMTP session.

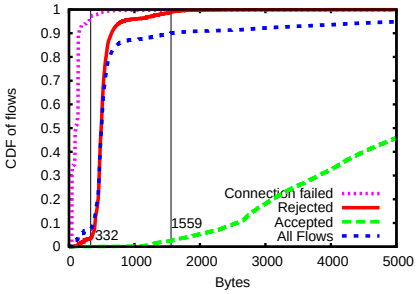


Figure 8.3: *The byte count distribution.*

	$x < 322$	$322 <= x <= 1559$	$x > 1559$
Failed	9302 (95.64%)	417 (4.28%)	7 (0.07%)
Rejected	11008 (3.59%)	409675 (96.66%)	3132 (0.74%)
Accepted	55 (0.09%)	1662 (2.74%)	58845 (97.16%)

Table 8.1: *The classification performance for x bytes per flow.*

[1]Note that only the traffic flowing from external SMTP clients to internal servers is considered

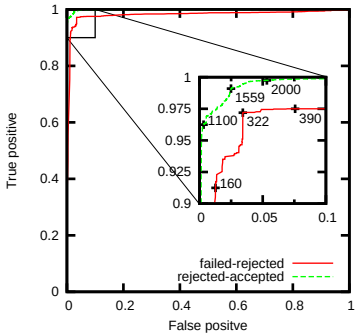


Figure 8.4: *The ROC curve for bytes per flow metric.*

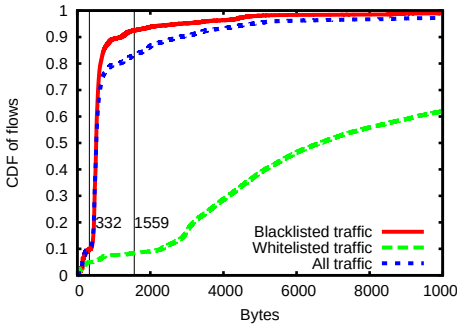


Figure 8.5: *The network-wide flow sizes.*

The CDF of byte counts for flows arriving at the mail server in week 1 is presented in Fig 8.3. The class of failed connections mainly consists of very small flows as only a small number of packets could be sent. 97% of these flows have less than 322 bytes. The size of most rejected flows is between 400 and 800 bytes. This corresponds to the size of the SMTP envelope. Lastly, the distribution of accepted flow sizes is dominated by the overall email size distribution and reaches 50% at around 5000 bytes. This is consistent with the findings of Gomes et al. [51]. The CDF for “all flows” in Fig. 8.3 is a superposition of the three classes weighted by their relative probability of appearance. All three classes are well visible in the total CDF even though it

is dominated by rejected flows due to the fact that around 80% of all flows are rejected.

Next, we determined two optimal threshold sizes to differentiate between *rejected*, *failed* and *accepted* flows. For this purpose, we constructed ROC curves [44] which plot the true positive versus the false positive rate of a detector for a range of thresholds. Fig. 8.4 shows the ROC curves for the detection of rejected vs. failed and accepted vs. rejected flows. The three classes are distinguishable with high precision. We selected the two thresholds 332 Bytes (rejected vs. failed) and 1559 Bytes because these points are closest to the top left corner and hence yield the best detection quality [44].

We evaluated the false positive rate of these threshold detectors on data of another week (week 2) and present the results in Table 8.1. The false detection rate is below 4.5% for all classes which is sufficiently accurate for the applications outlined in Section 8.4[2]. We also analyzed the power of other flow properties to discriminate between the three classes. In addition to *bytes per flow*, also *packets per flow* and *average bytes per packet* are well suited for this purpose [123].

It is important to note that packet sampling would affect our approach. Over 90% of the rejected SMTP sessions consist of 10 or less packets and more than 90% of accepted sessions have less than 40 packets. With a sampling rate of 1:100 or even 1:1000, the resulting flows would mostly consist of 1 or 2 packets. This would weaken the usefulness of the bytes and packets per flow metrics; yet, our analysis suggests that it could still be possible to distinguish between rejected and accepted flows using *average bytes per packet* [123]. Further, adaptive sampling techniques are being developed [113] that could perhaps address this problem also. We intend to look further into the issue of sampling in future work.

Network-wide characteristics

The classification of flows based on their size allows to passively monitor pre-filtering activity in large networks in a scalable manner, without resorting to server logs. To validate that the observed characteristics also hold on a network-wide scale, we show the characteristics of black- and whitelisted traffic for the 50 most active mail servers in our network in Fig. 8.5. The shape of black-/whitelisted curves nicely reflects the characteristics of re-

[2]The flow labels assigned by our system are to be treated mostly as “soft” labels. Further accuracy could be achieved by using e.g., clustering algorithms on additional flow fields.

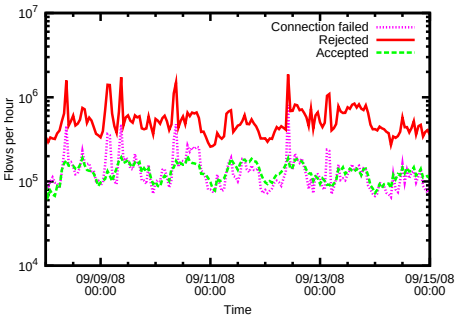


Figure 8.6: *The network-wide pre-filtering statistics.*

jected and accepted flows from Fig. 8.3. Hence, (i) the vast majority of traffic from blacklisted hosts is rejected by our network in pre-filtering and (ii) we are able to infer this reject decisions from flows sizes only. Individual server performance differences are addressed in detail in Section 8.4.1.

The generation of network-wide pre-filtering statistics, as illustrated in Fig. 8.6, allows to easily estimate the amount and track the dynamics of incoming spam at the ISP-level. An ISP is now able to investigate the root cause of anomalies in rejected/accepted traffic. Potential causes are global spam campaigns that are visible on many servers, spamming attacks targeted to a small set of servers or misconfiguration and performance problems of single servers.

8.4 Applications

We now turn our attention to potential applications of our method. In Section 8.4.1, we demonstrate how it can be used to passively analyze the configuration of mail servers and troubleshoot misconfigured servers. We then explore the feasibility and potential of a collaborative filtering system among the various mail servers in Section 8.4.2.

8.4.1 Email Server Behavior

Today, adequate configuration and maintenance of mail servers is a time-consuming process. It would be very helpful for operators to get a performance map of the various mail servers present in the network. The state of pre-filtering deployment in the network could be checked regularly and potential configuration problems, (e.g., the presence of open relay servers), could be addressed proactively.

To compare the pre-filtering performance of internal servers, we define the *acceptance ratio* of a server to be the number of accepted SMTP flows divided by the number of total SMTP flows seen by the server. A high ratio of, for example, 0.9 indicates that 90% of all incoming SMTP sessions are accepted, whereas a low ratio indicates that most of the connections are rejected during the TCP handshake or the SMTP envelope. Clearly, the observed acceptance ratio for a server is affected by two parameters: (i) the *traffic mix* of ham and spam for this server, and (ii) the server *prefiltering policy*. To address the former, we estimated the spam/ham mix ratio for each server with the help of the XBL blacklist from Spamhaus. Our analysis shows that spam (flows from blacklisted sources) is evenly distributed among servers. 81% of the servers have a spam load between 70% and 90%, consistent with [63]. This results implies that *big differences in servers' acceptance ratios cannot be attributed to different traffic mixes*.

The server policy issue is somewhat trickier. The above numbers imply that, if all servers were at least using a blacklist, the acceptance ratio of most internal servers should be between 0.1 and 0.3, with differences attributed to traffic mix and sophistication and/or aggressiveness of pre-filtering policies (e.g., greylisting, etc.). Instead, the acceptance ratios of the top 200 servers for week 3 of our data set range from 0.003 up to 0.93 with a mean of 0.33 as can bee seen in Fig. 8.7. 35% of the servers have an acceptance ratio > 0.30. Based on the above traffic mix estimation, we conclude that they are accepting a lot of traffic from spam sources. This could imply: (i) a regular server that is sub-optimally configured, lacks sophisticated or even simple pre-filtering measures (e.g., lack of time, caring, or knowhow), and/or should at least raise an eyebrow; or (ii) a server whose intended policy is to accept all messages (e.g., servers that apply content-based filtering only, honeypots, etc.)

To verify this assumption, we sent emails to all servers from two different IP addresses: an address blacklisted by Spamhaus and Spamcop and an

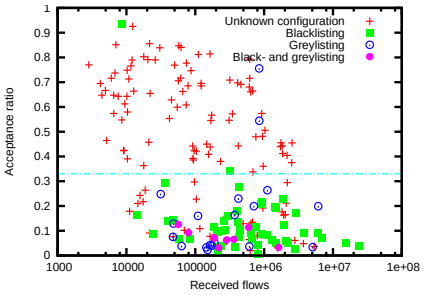


Figure 8.7: *The server acceptance ratios vs. traffic volume.*

ordinary, not blacklisted address[3]. The reaction of the servers to our sending attempts clarified whether the server was using greylisting and/or blacklisting. The servers classified as 'unknown' are those servers for which the reaction was not conclusive. The high concentration of black- and greylisting servers below the average ratio shows that, indeed, these servers implement basic pre-filtering techniques, whereas servers that do not implement them lie mostly above average. Also, with increasing volume (to the right), servers with high acceptance ratios tend to disappear. This affirms that administrators of high-volume servers (have to) rely on aggressive pre-filtering to master the flood of incoming mails. We also manually checked high acceptance servers and found no honeypots trying to deliberately attract and collect spam.

We conclude that differences in acceptance ratio are mainly due to configuration issues and that there is a large group of servers that might need a wake-up call or could profit from the expertise of other servers.

8.4.2 Collaborative Filtering

Given the above observations of unbalanced mail server performance, what could an ISP do to improve overall pre-filtering performance in its network? Surely, the ISP could contact individual administrators and organize a meeting to present the statistics about mail server performance where administrators would exchange their knowhow. However, this requires a lot of organizational effort, needs to take place regularly, and attendance of administrators

[3]It is important to stress that this, and other "manual" investigations we performed in this section are only done for validation purposes, and are not part of the proposed system.

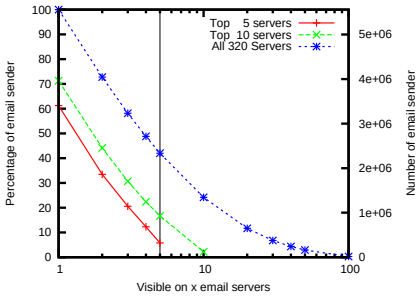


Figure 8.8: *The visibility of the email senders.*

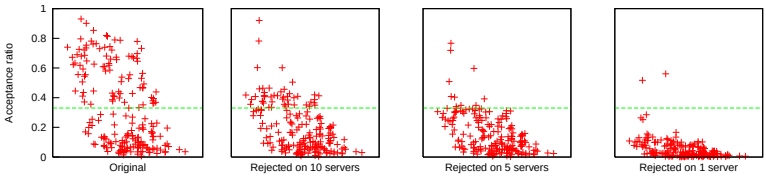


Figure 8.9: *The improvement potential when using collaborative filtering.*

is not guaranteed. Furthermore, educating customers' email administrators is usually not the ISP's business.

Therefore we investigate a passive way enabling *all* servers to profit from the local knowledge and the best existing anti-spam techniques already present in *some* servers in the network. By accepting more or less emails from a client, email servers actually perform an implicit rating of the very client. With our method, these ratings could be extracted from traffic and used to build a collaborative filtering system (CFS), or more properly, a collaborative rating/reputation system. The system would recommend accepting/rejecting mails from a client, based on the behavior of the collective of all servers: "This host was rejected by 80% of the servers in the network. Probably you should reject it as well."

It is important to note that the added value of the collected information depends on the ability of different servers to block different flows: heterogeneous knowledge or heterogeneous pre-filtering detection techniques and policies are *desirable* since this increases the chance that at least some methods will detect the spammer.

At the same time, a significant overlap of the sets of email senders visible to each server is needed, in order to achieve useful and trusted enough ratings. We analyzed the visibility of the sending hosts on three sets of top 5, top 10 and all internal servers (see Fig. 8.8). More than 5.5 million email senders are visible on the 320 internal servers. Moreover, 72% of all email senders are visible on at least two internal servers. This means that for 72% of the email senders a second option from another server is available. Note that even the top 5 email servers only see 61% of the sending hosts. In addition, only 6% of the email senders are visible on all of these five servers. This percentage is increased to 17% for the top 10 servers. By using information from all internal servers, 42% of the sending hosts are visible on at least 5 different servers, which is a promising foundation for building a CFS. In addition we explicitly analyzed the visibility of hosts that are listed in a black- or whitelist. The visibility of blacklisted hosts follows the overall host visibility. However, the visibility of whitelisted hosts is even better (e.g., 60% of the whitelisted hosts are visible on at least 5 different servers).

The actual implementation of a CFS is beyond the scope of this chapter. Nevertheless, we are interested here in estimating the potential of such a system by simulating simple filter rules. Specifically, we simulated an *online* version of the system where blocklists are created as incoming flows are parsed according to some simple rules. Specifically, counters are maintained for senders that have been consistently blocked with the following rules:[4] As soon as a se10nder's connections have been blocked by at least 10, 5 or 1 server(s), respectively, the sender is entered into our blocklist. Then, if a sender is on the list all incoming connections by this sender will be counted as rejected. The resulting acceptance ratios assuming all servers are using the CFS are shown in Fig 8.9.

There is an inherent tradeoff in the number of server "votes" used to make a decision. By requiring many rejecting servers (e.g., 10) for membership in the blocklist, the reliability of the filtering becomes quite strong (only 3 blocked hosts were actually whitelisted). Yet, the size of the blocklist is reduced, and provides less additional information than blacklists. Specifically, in the 10 server case, the blocklist has 83% overlap with the Spamhaus blacklist and could be used to block 47% of all SMTP sessions. In the other extreme, if only one rejecting server is required to be put on the blocklist, acceptance ratios of all servers are dramatically reduced. Further, the blocklist overlap with Spamhaus is only 70% and could be used to block up to 95%

[4] To reduce the effects of greylisting we delayed the blacklisting process by 15 min.

of all SMTP connections. However, 185 members were whitelisted. That is, requiring only one rejecting server introduces a higher false rejection rate. Nevertheless, it is important to note that in both cases, a significant amount of hosts identified by this method are *not* found in the blacklist, underlining the collaborative nature of the system, and implying that simply making under-performing servers use a blacklist, would not suffice.

Concluding, our estimation shows that there is a significant information overlap that can be leveraged to improve overall pre-filtering by making local server information accessible to the entire network, in an almost seamless manner. Today, pre-filtering is dominated by the use of DNSL blacklists. However, the CFS is independent of the very techniques applied and will automatically profit from upcoming improved techniques.

As a final note, due to the inherent "softness" of the labels attached by such a flow-based system, and the various risks of blocking a session in the network core, we stress here that our system is not intended as an actual blocking filter, but rather as a reputation rating system, which individual email servers can *opt* to include in their pre-filtering phase. Consequently, servers whose policy is to accept all emails, do not have to be affected by our system, although using it, could perhaps provide hints to the content-based filter.

8.5 Discussion

Although being only a first step, we believe the proposed method has important potential to be applied in production networks and also paves the way for future research in the area of network-level and network-wide characteristics of spam traffic. In this section we discuss some possible limitations of our approach.

Delay: The flow data is exported by the router only after the TCP connection is completed. Therefore, the information about the flow is delayed at least until the session is closed. We measured this delay to be less than 7.7 seconds for 90% of all SMTP flows. This illustrates that any application based on flow characteristics is limited to near-realtime. In particular, properties of a flow can not be used to intercept this very flow. For our proposed applications, this limitation is acceptable as we are not interested in using the properties of a flow to intercept this very flow, but rather subsequent ones.

Flow size manipulation: In principle, spammers could adapt to our method by prolonging the envelope phase, for instance by sending multiple RCPT

or HELO commands. This would indeed increase the number of transmitted bytes per flow but will, at the same time, increase the number of packets. However, the average number of bytes per packet remains small and the bytes per packet metric could still be used for the classification. Further, the spammer could try to increase the number of bytes transmitted in each command by using long email addresses, but the maximum size for a command is limited to 521 characters [75]. Moreover, any deviation from normal SMTP behavior could easily be detected and mail servers could enforce the shortness of the envelope phase as there is no need to be overly long in a normal use case. In addition, the misbehavior of the host could be published to make this information available for other servers.

In a more sophisticated scenario, a spammer could take over different internal hosts and reconfigure them as internal mail servers. By sending accepted SMTP traffic from bot to bot, one could try to positively influence the CFS ratings for these bots. The design of the CFS needs to be aware of this problem. In a first step, only servers that have been active over a longer time period (i.e., they have been accepting mails for at least several weeks) and get a certain amount of connections from trusted email servers (e.g., on a whitelist) could be included into the filtering process.

8.6 Summary

Mail server administrators are engaged in an arms race against spammers. They urgently need new approaches to fight state-of-the-art attachment spam increasingly originating from low-profile botnet spammers. In this work, we demonstrated that simple flow metrics, such as byte count, packet count, and bytes per packet, successfully discriminate between spam and ham flows when pre-filtering is deployed in mail servers. Thus, one could infer individual mail server's decisions with respect to the legitimacy and acceptance of a given connection. This allows an operator i) to concentrate dispersed mail server knowledge at the network core and ii) to passively accumulate network-wide spam statistics, profile filtering performance of servers, and rate clients. Thus, the advantages of flow and server log analysis finally meet at the network core. We believe this is an important step towards successfully fighting spammers at the network-level.

Chapter 9

Tracking Network Reachability

After the preceding case study on a mail services, we focus in this Chapter on service-independent techniques for connectivity tracking. The relevance of this topic is illustrated by the fact that even 20 years after the launch of the public Internet, operator forums are still full of reports about temporary non-reachability of complete networks. Therefore, we design a troubleshooting application that helps network operators to track connectivity problems occurring in remote autonomous systems, networks, and hosts. In contrast to existing solutions, our approach relies solely on flow-level information about observed traffic, is capable of online data processing, and is highly efficient in alerting only about those events that actually affect the studied network or its users.

9.1 Introduction

"Please try to reach my network 194.9.82.0/24 from your networks ... Kindly anyone assist", (NANOG mailing list [102], March 2008). Such e-mails manifest the need of tools that allow to monitor and troubleshoot connectivity and performance problems in the Internet. This particularly holds from the perspective of an individual network and its operators who want to be alerted about disrupted peerings or congested paths before customers complain.

Both researchers [69, 88, 161, 162] and industrial vendors [4, 19] have made proposals for detecting and troubleshooting events such as loss of reachability or performance degradation for traffic that they exchange with other external networks, unfortunately with mixed success. Predominantly, such tools rely on active measurements using ping, traceroute, etc. [88, 162]. Besides, researchers have suggested to leverage control plane information such as publicly available BGP feeds [66, 69, 106], although Bush et al. [18] point out the dangers of relying on control-plane information. Other concerns about existing tools include a high "dark" number of undetected events [66], a narrow evaluation solely in the context of a testbed or small system [106, 161], or the time gap between the occurrence of an event and its observation and detection [66].

In this chapter we propose FACT, a system that implements a **F**low-based **A**pproach for **C**onnectivity **T**racking. It helps network operators to monitor connectivity with *remote* autonomous systems (ASes), subnets, and hosts. Our approach relies on *flow-level* information about observed traffic (and not on control-plane data), is capable of *online data processing*, and *highly efficient* in alerting only about those events that *actually affect* the monitored network or its users.

In contrast to existing commercial solutions [4,19], we do not consider aggregate traffic volumes by interface or by peering to detect abnormal events, but pinpoint on a *per-flow basis* those cases where external hosts are unresponsive. On the one hand, this requires careful data processing to correctly handle asymmetric routing and to eliminate the impact of noise due to scanning, broken servers, late TCP resets, etc. On the other hand, our flow-based approach allows to compile accurate lists of unresponsive network addresses, which is a requirement for efficient troubleshooting.

To test our system we rely on an one-week flow-level trace from the border routers of a medium-sized ISP [143]. We demonstrate that our approach can be leveraged to detect serious connectivity problems and to summarize suspicious events for manual inspection by the network operator. Importantly, replaying flow traces from the past, FACT also reliably recognizes reported connectivity problems, but only if those are relevant from the perspective of the studied network and its users. Overall, we believe that our approach can be generally applied to small- to medium-sized ISPs, and enterprise networks. In particular networks that (partially) rely on default routes to reach the Internet can strongly benefit from our techniques, since they allow to identify critical events even if these are not visible in the control plane information.

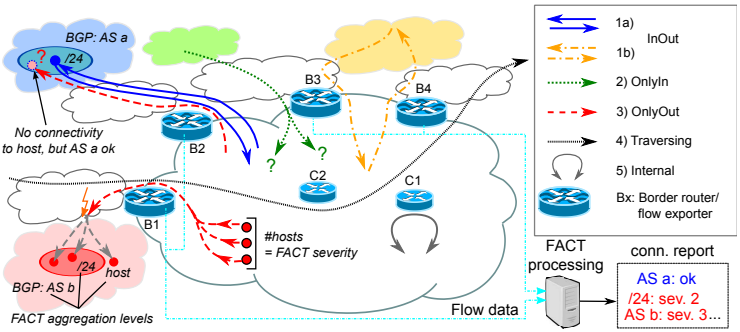


Figure 9.1: *The measurement infrastructure and flow types.*

9.2 Methodology

Our goal is to enable network operators to monitor whether remote hosts and networks are reachable from inside their networks or their customer networks, and to alert about existing connectivity problems. Such issues include cases where either we observe a significant number of unsuccessful connection attempts from inside the studied network(s) to a specific popular remote host, or where many remote hosts within external networks are unresponsive to connection attempts originated by potentially different internal hosts.

To obtain a network-centric view of connectivity, we rely on flow-level data exported by *all* border routers of a network, see Fig. 9.1. In this regard, our approach is generally applicable to all small- and medium-sized ISPs, and enterprise networks. Monitoring the complete unsampled traffic that crosses the border of our network allows to match outgoing with incoming flows and to check for abnormal changes in the balance between incoming and outgoing flows for external endpoints at different *aggregation levels* (hosts or networks). In particular networks that (partially) rely on default routes to reach the Internet can strongly benefit from such an approach, since it allows to identify critical events even if these are not visible in the control plane information.

As shown in Fig. 9.1, we distinguish between five *flow types*: `Internal` connections never cross the network border, and thus are neither recorded nor studied further in our approach. Since the scope of this work is limited to cases where remote hosts or networks are unresponsive to connection attempts originated by internal hosts, we ignore flows that traverse our network

(`Traversing`) or flows for which we cannot find traffic in the outbound direction (`OnlyIn`), e.g., caused by inbound scanning. If we can associate outgoing flows with incoming flows, we assume that external hosts are reachable (`InOut`) and also take this as a hint that there exists connectivity towards the remote network. Note that the incoming flow can enter the network via the same border router that was used by the outgoing flow to exit the network. Yet, due to the asymmetric nature of Internet paths this is not necessary [106]. Finally, we observe flows that exit the network but we fail to find a corresponding incoming response (`OnlyOut`).

To detect potential connectivity problems, we focus on the latter category `OnlyOut`. Note that we rely on the assumption that our measured flow data is complete, i.e., for any outgoing flow the associated incoming flow is observed by our collection infrastructure provided that there has been a response in reality. Evidently, network operators only want to get informed about critical events that include loss of connectivity towards complete networks or towards popular hosts that a significant share of internal hosts tries to reach. Our approach to achieve this goal is twofold.

First, we rely on data *aggregation* to investigate connectivity towards complete networks. More precisely, we aggregate occurrences of `OnlyOut` flow types across external hosts, /24 networks, or prefixes as observed in public BGP routing tables. For example, only if we observe within a certain time period a considerable number of `OnlyOut` flow types towards different hosts of a specific external network, and no `InOut` types, we conclude that the complete external network is currently not reachable for internal hosts. Hence, our decision is not based on observed connectivity between a single pair of internal and external hosts.

Second, we take into account the number of internal hosts that are affected by connectivity problems towards a host, network, or BGP prefix, i.e., the *severity of an observed event*. For example, loss of connectivity towards an individual external host is interesting for a network operator if a large number of different internal hosts fail to reach such a popular service. Moreover, knowing the number of affected internal hosts is crucial to extract short summaries of candidate events which network operators can check manually in reasonable time.

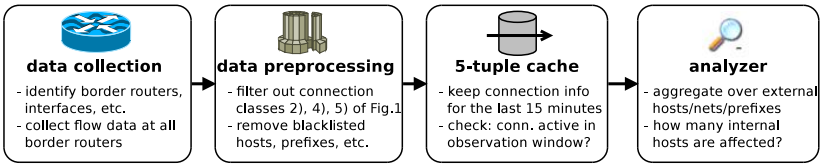


Figure 9.2: *The architectural components of FACT.*

9.3 Data Sets

We investigate our approach using our measurement setup based on data collected in the SWITCH network [143] (see in Section 2 for more details about the general measurement setup). For our studies we have collected a trace in September 2010 (`OneWeek`) that spans 7 days and contains unsampled NetFlows summarizing all traffic crossing the 6 border routers of the SWITCH network. This results in $14k - 40k$ NetFlow records per second. In addition to `OneWeek` we extract some shorter traces to study selected connectivity problems from the past, see Section 9.5.

9.4 Connectivity Analysis

The implementation of FACT includes four major components, see Fig. 9.2. After `data collection`, a `preprocessing` step removes some flows from the data stream, e.g., blacklisted hosts or information that is not needed to achieve our goals. For a limited time we keep the remaining flows in the `5-tuple cache`, which is continuously updated with the latest flow information. In the following we will provide more details about the implementation of the individual components.

9.4.1 Data Collection and Preprocessing

In addition to standard flow information including IP addresses, port numbers, protocol number, packet counts, byte counts, etc., we store identifiers for the border routers and interfaces over which traffic with external networks is exchanged. Next, we exclude a considerable number of unnecessary flows to save memory and computational resources, but also eliminate flows that have turned out to be harmful for the detection of connectivity problems.

Such flows include for example traffic from/to PlanetLab hosts or bogon IP addresses, and multicast. For now, we generate an appropriate blacklist manually, but we plan to automate this process in the future. For reasons already described in the preceding section, we remove in this step also all flows of the class `Traversing` and `Internal`, see Fig. 9.1.

9.4.2 5-tuple cache

The subsequent data processing respects the fact that the active timeout of our flow collection infrastructure is set to 5 minutes.[1] Therefore, we partition the timeline into intervals of 5 minutes and proceed with our data processing whenever such a time interval has expired. Our goal is to maintain for each interval a hash-like data structure (*5-tuple cache*) that, for observed flows identified by IP addresses, protocol number, and application ports, stores and updates information that is relevant for further analysis. This includes packet counts, byte counts, information about the used border router and the time when the flows were active for the in and out flow. Note that at this point we implicitly merge unidirectional to bidirectional flows (biflows).

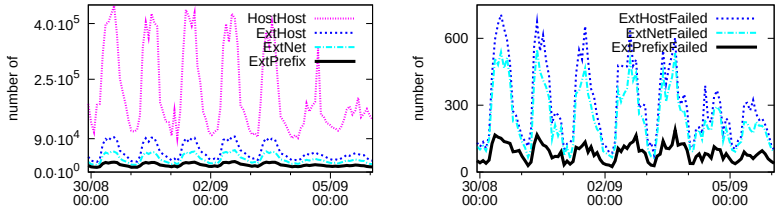
After the time interval has expired we extract from the obtained biflows and remaining unidirectional flows two sets: The set `ConnSuccess` includes those biflows of type *InOut* where at least one of the underlying unidirectional flows starts or ends within the currently studied time interval and are initiated by internal hosts[2]. The second set, called `ConnFailed`, includes only those unidirectional flows of type `OnlyOut` where the outgoing flow either starts or ends in the currently studied time interval. To reduce the effect of delayed packets (e.g., TCP resets), we here ignore unidirectional flows if a corresponding reverse flow has been observed during any of the preceding time intervals.[3] All other flows of the *5-tuple cache* that are not in the set `ConnSuccess` or `ConnFailed` are excluded from further consideration for this time interval.

While `ConnSuccess` flows indicate that an internal host in our network can indeed reach the external host, we take occurrences of `ConnFailed` as a hint for potential connectivity problems with the remote host. However, the latter assumption does not necessarily hold when applications (e.g., NTP

[1] After 5 minutes even still active flows are exported to our central flow repository.

[2] We rely on port numbers to determine who initiates a biflow.

[3] Our hash-like data structure is not deleted after a time period of 5 minutes but continuously updated. Only if a biflow is inactive for more than 900 seconds, it is removed from our hash-like data structure.



(a) Visible external destinations. (b) Unresponsive external destinations.

Figure 9.3: *The number of external hosts, networks, and prefixes.*

or multicast) are inherently unidirectional. Hence, we exclusively take into account HTTP traffic using port 80, which is symmetric by nature and due to its popularity visible in any type of network.[4] More marginal fine-tuning of our data processing is required. Yet, given space limitations we refrain from providing more details.

9.4.3 Analyzer

To study observed connectivity with remote hosts and to detect potential problems, the analyzer component processes the sets `ConnSuccess` and `ConnFailed` every 5 minutes. both We aggregate `ConnFailed` and `ConnSuccess` flows for the same pair of internal and external host if we find more than one flow, possibly with different port numbers. The obtained host-host tuples are classified as `HostHostSuccess` if at least one `ConnSuccess` flow has been identified, `HostHostFailed` otherwise. Based on this initial aggregation step, we independently compute three stronger aggregation levels: we group host-host tuples into one tuple if they affect the same external host (`ExtHostSuccess` or `ExtHostFailed`), the same external /24 network (`ExtNetSuccess` or `ExtNetFailed`), and BGP prefixes (`ExtPrefixSuccess` or `ExtPrefixFailed`). With respect to the last granularity, we use publicly available BGP routing tables to determine the corresponding BGP prefix for a given external host. Again, we classify an aggregate class as `Success` if at least one tuple is marked as `HostHostSuccess`.

Fig. 9.3 displays the number of visible and unresponsive external destinations if the three aggregation granularities are applied to `OneWeek`, see

[4] Experiments relying on DNS traffic turned out to work as well.

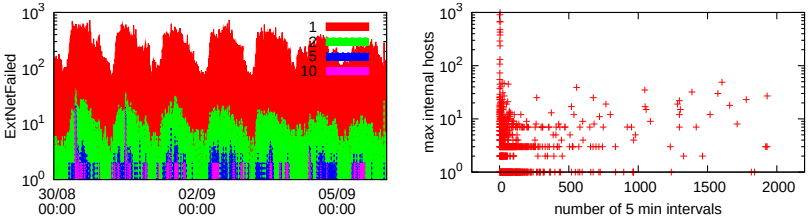
Section 9.3. According to Fig. 9.3(a) the absolute number of visible external destinations shows a strong daily and weekly pattern irrespective of the used aggregation level. Aggregating from host-host into `ExtHostFailed` and `ExtHostSuccess`, respectively, reduces the peaks from 525*K* to 90*K* tuples (/24s: 50*K*, prefixes: 25*K*). This provides evidence for the high visibility that our data has on external networks. However, Fig. 9.3(b) reveals that generally only a small fraction of external hosts (peaks of 700) are unresponsive and therefore classified as `ExtHostFailed` according to our methodology. This fraction is significantly smaller for `ExtNetFailed` (peaks of 600) and `ExtPrefixFailed` (peaks of 180), respectively.

However, to cope with daily and weekly fluctuations and to limit the degree to which a single internal host (e.g., a scanning host) can impact our connectivity analysis, we need to take into account the *severity of an observed event* as well. By this we understand the number of internal users that actually fail to establish connectivity with a specific external host, /24 network, or BGP prefix during our 5 minute time intervals. Figure 9.4(a) displays the number of external /24 networks that are unresponsive to 1, 2, 5, and 10 internal hosts for the time spanned by `OneWeek`. The majority of these `ExtHostFailed` "events", namely 98%, only affect 1 internal host.

Yet, here it is important to study Fig. 9.4(b). It is also based on `OneWeek` and counts for every external host the number of 5-minute time intervals for which it has been classified as `ExtHostFailed`. This number (x-axis) is plotted against the maximum number of internal hosts (y-axis) that failed to establish connectivity with this external host (`ExtHostFailed`) at *any* 5-minute interval of `OneWeek`. We find that the majority of external hosts (96%) are only unresponsive in less than 10 time intervals of our trace. However, some hosts are unresponsive most of the time, e.g., abandoned ad servers. Data preprocessing as described in Section 9.4.1 could be refined to automatically blacklist such hosts and possibly their networks. Finally, we observe few external hosts that are unresponsive only during a small number of time intervals, but with a high maximum number of affected internal hosts. Cross-checking with technical forums in the Internet, we find that these events include for example a Facebook outage on August 31, 2010.

We point out that the data processing in FACT is faster than real time for SWITCH, a medium-sized ISP covering an estimated 6% of the Internet traffic in Switzerland and approximately 2.2 million IP addresses: flow data spanning 5 minutes[5] can be processed using a single thread in less than

[5] We see up to 200 million flows per hour.



(a) Number of `ExtNetFailed` that are unresponsive to 1, 2, 5, and 10 internal hosts.

(b) Frequency analysis of `ExtHostFailed` events

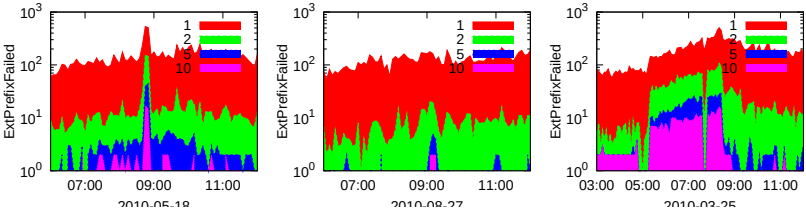
Figure 9.4: *The severity of observed events.*

three minutes with a maximum memory consumption of less than 4GB. Aging mechanisms for our data structures[3] ensure that the overall memory consumption does not increase during long-term use of our system. Due to hash-like data structures we can access individual flows in our 5-tuple cache in constant time. The total time required for data processing mainly depends on the number of active flows. In principle, it is even possible to parallelize our processing by distributing the reachability analysis for different external networks to different CPU cores or physical machines. Yet, we leave it to future work to study FACT's performance for large tier-1 ISPs and how to make it robust against potentially higher false positive rates if sampled flow data is used.

9.5 Case Studies

In this section we present a short analysis of three connectivity problems that were either detected by the network operator or publicly documented. To analyze those cases, we rely on data collected as discussed in Section 9.3.

Black-holing: On May 18, 2010, all services in an external /24 network were not accessible from SWITCH between 08:30 and 08:45. According to the operators of SWITCH, this problem was most likely due to a tier-1 provider that black-holed parts of the reverse traffic towards SWITCH. Yet, at this time the operators could only speculate how many hosts and customers, or even other /24 networks were affected by this problem. Applying FACT we confirm that the reported /24 network is indeed reported as unreachable at around 08:30. Surprisingly, FACT reveals that the overall number of unreach-



(a) Black-holing (b) RIPE/Duke event (c) Partitioned IXP

Figure 9.5: *A case studies: unresponsive BGP prefixes*

able hosts and /24 networks has doubled compared to the time before 08:30 while the number of unresponsive BGP prefixes is increased by a factor of even 6, see Fig. 9.5(a). Moreover, the reported /24 network is not even in the top ten list of the most popular unresponsive networks. This suggests that the impact of this event has been more serious than previously believed.

RIPE/Duke event: On August 27, 2010, some parts of the Internet became disconnected for some 30 minutes due to an experiment with new BGP attributes by RIPE and Duke University [117]. FACT reveals that at around 08:45 the number of popular unresponsive /24 networks indeed doubled. According to Fig. 9.5(b), for some BGP prefixes more than 15 internal hosts failed to establish connectivity. Yet, overall our analysis reveals that the impact of this incident on SWITCH and its customers was quite limited compared to the public attention that this event obtained.

Partitioned IXP: After scheduled maintenance by AMS-IX, the SWITCH's connection to that exchange point came back with only partial connectivity. Some next-hops learned via the route servers weren't reachable, creating black holes. The next morning, several customers complained about external services being unreachable. Overall, it took more than four hours until the problem was finally solved by resetting a port. Fig. 9.5(c) shows that the number of unresponsive BGP prefixes is almost ten times higher than normal, over a time period of more than four hours. We believe that FACT would have helped to detect such a serious problem much faster and provided valuable hints about the origin of the problem.

9.6 Related Work

Approaches for detecting and troubleshooting reachability problems can be generally classified into two classes: *active probing* and *control plane based*.

With respect to *active probing*, Paxson et al. [106] are probably the pioneers to use traceroute for studying end-to-end connectivity between a (limited) set of Internet sites. Zhang et al. [162] perform collaborative probing launched from Planetlab hosts to diagnose routing event failures. Commercial solutions such as NetQoS [19] or Peakflow [4] generally rely on *active* measurements using ping, traceroutes, or continuous SNMP queries to network devices. Moreover, they frequently aggregate traffic volumes per interface, peering links, etc. to detect abnormal events, and hence do not base their analysis on a flow-level granularity as our work suggests. In contrast to active probing, the passive monitoring approach of FACT does not impose any traffic overhead and, importantly, only creates alerts for those unreachable hosts/networks that users actually want to access. Finally, FACT avoids an intrinsic problem of active probing techniques such as ping or traceroute, namely the implicit assumption that reachable hosts actually do respond to such tools.

In addition to active probing, a considerable number of research papers, e.g., [46, 66] rely almost exclusively on *control-plane information* in the form of BGP routing feeds. However, Bush et al. [18] have clearly pointed out the dangers of such an approach, e.g., the wide-spread existence of default routes. In contrast, FACT is able to detect unreachability at multiple and finer granularities (e.g., on a host basis) than any approach that is purely based on routing data. Later work including e.g., Hubble [69] and iPlane [88] rely on hybrid approaches combining active measurements with BGP routing information. Feamster et al. [45] adopt such an approach to measure the effects of Internet path faults on reactive routing. Overall, we believe that the passive approach adopted by FACT is very powerful compared to active probing and control-plane based techniques. Yet, we plan to integrate active probing into our system to crosscheck detected reachability problems and to pinpoint the underlying causes.

9.7 Summary

We have proposed FACT, an online data processing system that helps operators to acquire facts about connectivity problems with remote autonomous

systems, subnets, and hosts. In contrast to existing solutions, our approach relies solely on flow-level information extracted from traffic crossing the border of the network. We showed, with the help of reported real-world events, that FACT can be used to alert only about those events that actually affect the studied network or its users. Importantly, data processing of FACT is already faster than real time for a medium-sized ISP.

Chapter 10

Conclusion

In this chapter we first provide a summary of our contributions. Then we analyze possible shortcomings and weaknesses of our work, and discuss how these open research issues can be addressed by future work. Finally, we complete this thesis by presenting the publications on which this work is based.

10.1 Contributions

In this thesis, we demonstrated that aggregating and analyzing flow sets across dimensions such as time, address space, or users provides additional information that is often overlooked. We used this information to widen the class of applications that can benefit from flow-based network measurements in the context of network monitoring and service troubleshooting. Along the three phases of preprocessing, annotation, and troubleshooting the achievements of the thesis at hand can be summarized as follows:

Fast Flow Processing Processing millions of flows within limited time to deliver statistics, reports, or on-line alerts to the user is challenging. For these tasks, we implemented FlowBox, a modular toolbox for online flow processing, which exploits concurrency to speed up data mining. FlowBox provides an interface written in Ruby suitable for rapid prototyping and exploits concurrency to speed up data processing. A basic byte counter application implemented in Ruby was able to process up to 500k flows per second on off-the-shelf hardware. FlowBox is

released to the community under GNU Lesser General Public License and is already used by further researchers and engineers for measurement applications.

Service Extraction Many monitoring applications use, unconsciously, SeSs as basic building blocks in their data processing. However, only limited work is done in the field of identifying these fundamental data structures in flow data. We contribute a service-agnostic, data driven approach to extract SeSs in real-time for unreliable passive large flow data. Our approach requires neither TCP flags nor precise flow time stamps, does not rely on port numbers and is able to cope with background radiation caused by mis-configured or malicious hosts probing the Internet. Our implementation is based on FlowBox and was executed on off-the-shelf hardware. It allows on-line dissection of large-scale network traffic with peak rates of more than 20 Gbit/s into services structures.

Characterizing the Service Landscape To understand the service landscape of the Internet, we first use service extraction to identify SeSs. Then, we study where network services are located, what traffic characteristics they reveal, and how characteristics change over time. One insight is that the deployment of end-user services running on high ports (such as Skype services) is more prevalent than widely thought (30% in 2011). Further, we perform a frequent itemset analysis of the spatial characteristics of detected services. We are able to identify service collocation patterns that can be exploited for service classification (e.g., Skype on T80-T443-TWIN or AOL Instant Messenger on T443-TCP-H).

Service Classification Many network operators want to annotate network services with traffic labels (e.g., Bittorrent, WebMail, etc.). We contribute flow-level techniques that can be used to separate webmail traffic from other HTTPS traffic. The novelty of our work goes into two main directions: (i) we leverage correlations across (related) protocols (e.g. IMAP, POP, SMTP, and webmail) and among hosts sharing a similar client base, and (ii) we identify and exploit timing characteristics of webmail applications. Based on these, we have introduced novel features, investigated their efficiency on a large flow data set, and used them to produce preliminary classification results on internal and external HTTPS servers. This is the first work to show that it is possible to uncover HTTPS webmail applications solely based on flow-level

data with approximately 93% accuracy and 79% precision. Although we limit our study to the labeling of webmail services such as Outlook Web Access, Horde, or Gmail, our approach is promising also for other classes of network applications.

Hybrid Classification We study how to augment service classification techniques such that they can be applied on very large-scale networks with reasonable efforts. To this end, we analyze if service labels from a subpart of the network can be projected to the overall network. Our idea is based on the observation that a service label for a given server socket is stable across multiple flows and different users accessing the socket. We exploit this fact to develop new labeling methods that reduce the overall collection and processing efforts. Our analysis showed that significantly more than 60% of the overall network traffic can be enhanced with traffic labels even if the known service labels have been derived from network traffic covering only 10% of the overall traffic.

Tracking Mail Servers After describing our annotation techniques, we turn to the first troubleshooting application. As an example, we choose e-mail as network application. We discuss how flow based information collected at the network core can be instrumented to counter the increasing sophistication of spammers and to support mail administrators in troubleshooting their mail services. In more detail, we show how the local intelligence of mail servers can be gathered and correlated passively, scalably, and with low-processing cost at the ISP-level providing valuable network-wide information. First, we use a large network flow trace from a major national ISP, to demonstrate that the pre-filtering decisions and thus spammer-related knowledge of individual mail servers can be easily and accurately tracked and combined at the flow level. Then, we argue that such aggregated knowledge does not only allow ISPs to monitor remotely what their "own" servers are doing, but also to develop new methods for fighting spam.

Tracking Connectivity Finally we propose techniques for connectivity tracking of network services. More than 20 years after the launch of the public Internet, operator forums are still full of reports about temporary reachability of complete networks. Therefore, we design a troubleshooting application that helps network operators to track connectivity problems occurring in remote autonomous systems, networks, and hosts. In contrast to existing solutions, our approach relies solely on

flow-level information about observed traffic, is capable of online data processing, and is highly efficient in alerting only about those events that actually affect the studied network or its users.

10.2 Critical Assessment

”Have no fear of perfection - you’ll never reach it.” This timeless quote of Salvador Dalí certainty applies as well on this work.

In general, a major challenge for any measurement system is scalability. Due to continuous advances in communication technologies, the available transmission bandwidth is constantly increasing. At the same time, the demand of end users is likely to grow even more. Therefore, network professionals are constantly concerned if the methods and approaches used today will still work in the mid- and long-term. Typically, this concern is reflected in questions like ”Does this still work, in 10 years?” or ”Would this work if the network was 10 times larger?”.

To discuss such questions, we distinguish between two scenarios. First, we have to accept that troubleshooting applications need to cope with drastically increased traffic volumes under the assumption that the granularity of the measurement data stays unchanged. For example, this will impose higher flow traffic rates for our measurement methods. Second, we discuss the situation where the metering system itself is overwhelmed by the amount of traffic. More precisely, this means that off-the-shelf flow meters (e.g., in a router) are unable to record all packets.

Coping with increased flow rate

We state that the presented troubleshoot applications will still work if the incoming flow rate is increased by a factor of 10 or 100. We argue that to achieve higher processing rates we can distribute the flow processing over multiple processing nodes. As discussed in Chapter 3, FlowBox supports this operation model by design.

For example, to scale up FACT presented in Chapter 9 we distribute the filtering and analysis over different hosts as Figure 10.1 illustrates. The scalability is achieved by allocating multiple parser and filter instances that process the incoming flow data in parallel. As a simple rule we can assign each flow exporter to one processing host. However, the 5-tuple cache and analyzer component of the FACT system require more work. After all, it is necessary

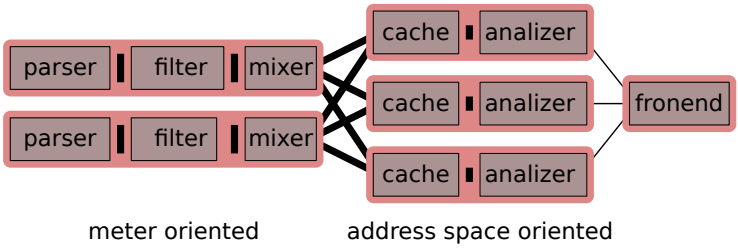


Figure 10.1: *Scaling FACT to work with any flow rate.*

to study *all* flows of a certain address space at the same component. Therefore, we have to partition the IP address space. More precisely, we have to make a certain processing node responsible to evaluate the connectivity status toward a certain number of BGP Prefixes. The filtering hosts simply need to parse IP addresses in the flow data and forward the data to the responsible processing host. Finally, we have to collect the connectivity status of the different analyzers and present the results. With the sketched approach, FACT can scale with the number of available processing nodes. Similar approaches are also feasible for the other techniques presented in this thesis.

Coping with restricted information

In the last section, we discussed how data mining can be scaled to process very high flow rates. The inherent assumption is that the flow meters are able to process the incoming data stream. If this is not the case, the operator has to invest into a new costly, dedicated monitoring infrastructure. This problem corresponds to the starting point for our PARS PRO TOTO approach discussed in Chapter 7. There, we used time-space sampling to reduce the overall collection and processing efforts.

Current developments in the field of Software Defined Networks (SDN) and the availability of off-the-shelf hardware such as OpenFlow [104] switches can help to efficiently implement time-space sampling, as it is widely used in this thesis.

For example, we can modify FACT to iteratively walk over the IP address space and evaluate the connectivity status for a limited number of BGP prefixes per time slot. In general, we believe that SDN techniques such as OpenFlow can help to implement lightweight network monitoring applica-

tions based on dynamic space-time sampling.

10.3 Future Work

”To every answer you can find a new question”. This yiddish proverb pointedly summarizes our experience. During our work, we discovered many interesting topics that should be covered by future work including research and engineers challenges.

Server Sockets

In this thesis we used SeSs to structure flow data and discussed in Chapter 5 some general characteristics including popularity and reachability. As direct future work, we should investigate how this information can be included into FACT to automate the current blacklist approach and improve detectability of events. In addition, more work is required to study the server/service landscape of the Internet using SeS. We believe this will further help to identify unused information that will foster new network measurement research and improve network management applications.

Space-Time Sampling

As discussed in the last section, Software Defined Networks equipment opens new ways of measuring large scale networks. With our work about enriching service labeling with dynamic time-space sampling we only touched the top of the iceberg of an new interesting research area: Exploiting dynamic space time sampling by SDNs to implement lightweight network monitoring applications.

Distributed Stream Processing

We believe that FlowBox provides an interesting playground to dive deeper into the problem of distributed stream processing of large scale network monitoring data. It allows researchers to develop and test troubleshooting applications that scale. However, there exist still many open questions that need to be addressed. For example, what is the optimal strategy to distribute the load across the different processing nodes. We will need a mechanism that dynamically balances the load across the processing nodes depending on the

currently observed traffic characteristics. This load balancing should try to reduce the number of hosts required to process the data to reduce the overall energy consumption of the metering system. This is an area that is of high interest for network professionals.

Ready to use

The ease of use of an application is crucial. Our troubleshooting application FACT is still in prototype state. The primary goal is to study the research problem of tracking the reachability of host, networks, or BGP prefixes. To obtain a ready-to-use application, a graphical user interface needs to be provided, for example a web interface to configure or visualize the findings of FACT.

Appendix A

DPI Labels

The DPI appliance is configured to collect the following information:

field	description
flow id	unique flow identifier
time first ms	system time when the fist packet was observed
time last ms	system time when the last packet was observed
packets	number of packets observed within this flow
bytes	number of bytes observed within this flow
application id	application identifier
l4 port src	TCP/UDP src port
l4 port dst	TCP/UDP dst port
server	server flag (1 = src, 2 = dst)
addr src	IP address of the sourc
addr dst	IP address of the destination

An (anonymized) example output:

```
...
6063967,1307484270121,1307484329000,1,97,32,4546,53,17,2,A.B.Y.X,A.B.Y.X,
6069492,1307484272453,1307484329000,1,273,32,53,52434,17,1,A.B.Y.X,A.B.Y.X,
6063967,1307484270131,1307484329000,1,224,32,53,4546,17,1,A.B.Y.X,A.B.Y.X,
6069235,1307484272465,1307484329000,1,272,32,53,17578,17,1,A.B.Y.X,A.B.Y.X,
6064029,1307484270133,1307484329000,1,90,137,3073,123,17,2,A.B.Y.X,A.B.Y.X,
6069539,1307484272481,1307484329000,1,95,32,30447,53,17,2,A.B.Y.X,A.B.Y.X,
6064029,1307484270133,1307484329000,1,90,137,123,3073,17,1,A.B.Y.X,A.B.Y.X,
6069591,1307484272497,1307484329000,1,90,137,123,123,17,2,A.B.Y.X,A.B.Y.X,
6065019,1307484270539,1307484329000,1,90,137,123,123,17,2,A.B.Y.X,A.B.Y.X,
6069591,1307484272497,1307484329000,1,90,137,123,123,17,1,A.B.Y.X,A.B.Y.X,
...
```

Appendix B

FlowBox

B.1 Sample Application

In this Section we provide more details about the implementation of FlowBox. Further we demonstrate the ease of use of FlowBox by crafting a sample application using less than 50 lines of Ruby code that counts number of bytes exchanged over the network

B.1.1 Example: Byte Counter Application

As discussed FlowBox provides the functionality to process flow data using only Ruby scripts. In this section we will use this feature to implement an application that counts the number of bytes which are exchanged over the network. As shown in Listing B.1 the implementation of this application requires less than 50 lines of code. This application consists of two processing units, namely the flow parser unit that reads the flows from disk (line 7) and a bytes counter unit that accumulates the required statistics (line 15). In addition the main thread (line 32) is waiting within a while loop on the end of the processing and periodically write the current statistics to the console (line 41).

The flow parser unit is a standard unit using a C++ extension to read the flows from a compressed bz2 file and to store the flow in a FlowContainer. The container is then forwarded to counter unit using the 'buffer_reader_bytecount' FIFO buffer (line 4). The byte counting is implemented with the help of the

```
#!/usr/bin/env ruby
require 'flowbox'
# buffer between reader and byte counter
buffer_reader_bytecount = FlowBox::Core::FlowContainerBuffer.new()

# prepare the reader
reader = FlowBox::Driver::Console::Parser.new()
reader.configure(
  :csgfile_input => ARGV[0], # pointer to the flow files
  :output => buffer_reader_bytecount,
  ...
)
reader.start()

# prepare the byte counter
perflow = FlowBox::Core::Flowblock.new()
perflow.configure(
  :input => buffer_reader_bytecount,
  :output => 'NULL' #  => 'NULL' End of Chain
)
# do stuff per flow in Ruby
bytes_c = 0
t1 = Thread.new do
  perflow.each do |flow|
    bytes_c += flow.bytes()
  end
end

time_last = Time.now.to_i
bytes_last = 0

while reader.finished() == false or t1.alive?
  sleep(10)

  time_now = Time.now.to_i
  bytes_now = bytes_c

  duration = time_now - time_last
  bytes = bytes_now - bytes_last

  puts "BYTES_PER_S, #{Time.now.to_i}, #{bytes/duration}"

  time_last = time_now
  bytes_last = bytes_now
end
```

Listing B.1: *A simple byte counter application using the prototyping unit*

class the FlowBlock. This class allows users to register a 'block' of code that is called for each flow within a container. A feature that is often used in the Ruby language to iterate in an user-friendly form over enumerative objects such as Arrays, Hashes, or Strings. In this application our code block (line

25) is called for each flow within a flow container and the total number of bytes is stored in the variable 'bytes_c'. The main thread is accessing this variable periodically (line 39) to calculate the required statistics.

the application can be modified to count only the bytes from TCP flows by solely changing the statement on line 25 to "bytes_c += flow.bytes() if flow.protocol() == 17" requiring no recompilation of the application.

B.2 Extended Performance Evaluation

In this section we evaluate the performance of the FlowBox using two different scenarios. The first scenario focus on the performance impact of choosing Ruby for flow processing compared to pure C++. By comparing different variations of our byte counter application introduced in Section B.1.1 we demonstrate that the overall flow throughput is not reduced by using interpreted Ruby scripts. The second scenario is based on a ring topology where the end of the processing chain is looped back to its begin. This ring topology allows us to evaluate the performance of FlowBox under very high flow rates that can't be achieve by replying real network traces from disk. Since flow replying is typically limited to the I/O performance of the used storage system and decompression of this data. Our analysis indicates, that the throughput in this scenario is mainly limited by the main memory I/O performance and not the interprocess communication.

B.2.1 Setup

To collect the performance measurements we used an off-the-shelf server equipped with four Quad-Core AMD Opteron 8350 HE processor, providing 64GByte main memory, and running a Debian Linux 2.6.30. We complied FlowBox using gcc 4.3.2 and and used Ruby Interpreter 1.9.2p290 to run the scripts. The flow throughput is estimated using two one hour NetFlow V9 traces that were collected at *midnight* and *midday* 2nd of April, 2012 on the border of the SWITCH network (see Section 2.1) consisting of 119.9 mio and 330.6 mio flows as summarized in Table B.1. The traces are compressed with bz2 and are stored on a local RAID 6 storage[1] requiring 4.3 GBytes. The decompression of the *midday* trace using command line tool bzip2 requires more than 25 minutes and requires 16 GByte disk space.

[1] Areca ARC-1220, 500GB, SATA II, 7'200 rpm

	midnight	*midday*
flows	119.9 Mio	330.6 Mio
file size	5.6 GByte	16.0 GByte
file size bz2	1.4 GByte	4.3 GByte
compression ratio	4.0	3.7
decompression time	438s	1341 s

Table B.1: *Flow traces used for the evaluation*

B.2.2 Impact using Ruby Interpreter

To evaluate the impact using Ruby for flow processing compared to pure C++ we transformed the byte counter application of Section B.1.1 into four different sub-programs. While the sub-program *reader_cc* and *counter_cc* are written in C++, the *reader_ruby* and *counter_ruby* sub-programs are Ruby scripts. The reader applications only consist of a flow parser, that reads the flows from the bz2 compressed files, without doing any further processing. Those sub-programs are used to estimate the additional cost caused by the byte counter functionality.

	midnight			midday		
case	real [s]	cpu [s]	load	real [s]	cpu [s]	load
reader_cc	220	436	198 %	786	1386	176 %
reader_ruby	216	432	200 %	781	1418	181 %
counter_cc	223	441	197 %	789	1412	178 %
counter_ruby	219	612	279 %	778	1875	241 %

Table B.2: *Performance Metrics*

For each sub-program we measured the running time (real) and the number of CPU-seconds used by the system on behalf of the process (cpu) required to process the trace *midnight* and *midday*. Base on this the percentage of the CPU that this job got (load) was calculated. To leverage variations caused by background jobs we repeated the runes five times and present the median of the correspond metrics in Table B.2.

We asked, first, whether using Ruby scripts slows down the flow processing. In particular, the *reader_ruby* and the *counter_ruby* sub-programs should require a longer running time than *reader_cc* and *counter_cc*. The answer is no, they don't. As Table B.2 reveals, all sub-programs require around 220

seconds to process *midnight* and around 780 seconds to *midday*. Actually, the ruby scripts run slightly faster than the C++ sub-programs. However the difference is within the standard deviation of 12 seconds and therefore not significant. This illustrates that relying on ruby to perfome flow processing does not impact the overall processing performance.

Section 3.1.2, the ruby based counter sub-program *counter_ruby* should be slower than its C++ counterpart since the code needs to be interpreted at runtime. The answer is yes. The total number of CPU-seconds used by the system on behalf of the process (cpu) is indeed significantly higher. As Table B.2 reveals, *counter_ruby* requires 279 while *counter_cc* requires only 176 seconds. This shows that total load is increased by almost 30%. Nevertheless the total running time is not affected, since this work was done in parallel with the reader. This illustrates the benefit of using independent worker units working in parallel.

reader_cc and *counter_cc* rather similar. This indicates that most of the processing time is spent for reading flows. This raises the question if our flow reader performs less well than expected. The answer is no. As Table B.1 reveals, the pure decompression of *midnight* using bzip2 requires 438 seconds. In contrary, the systems requires only 220 seconds to execute *reader_cc*, corresponding to a 60% reduced running time. This good run time behavior of FlowBox is achieved by parallelizing the decompression using multiple threads. In addition, the system spend only 436 CPU-seconds on behalf of the *reader_cc*. This is equivalent with the time system spend to execute the decompression. Illustrating that the actual parsing of the flows requires only a few cpu cycles compared to the decompression.

This opens the question if we should further try to optimize the reader component. The answer is no. First, a further parallelization of the decompression clearly increase the load on the the general I/0 bus including disk and main memory. This can easily result into situation where the total throughput of system is deceased due to cache thrashing. But most likely, as long no SSD storage is used, all improving efforts will be limited by disk I/O first. Second more important, in operative networks flow meters normally export uncompressed measurement data over the network and therefore this disk I/O or compression limitations do not exist. Instead, we propose distribute the reader components over different physical hosts in a data center and run multiple processing chains in parallel to scale Flow Box as discussed in Section3.1.1 .

B.2.3 Ring Topology

In this section we try to asses the performance of the FlowBox core by masking possible limitation caused by I/O devices like reading files from compressed files or network capturing.

In fact we chain n standard units together into a ring topology by connecting the end of the processing chain with its begin. Each unit iterates over the received flow containers and sums up the number of bytes of each flow. At the beginning $2*n$ FlowContainers were injected into ring topology. After 15 seconds we stopped the processing and measured the number of CPU-seconds the application spent in user and kernel time. In addition we recorded the number of flows that were processed by the first unit. Again to leverage variations caused by background jobs we repeated the runes 20 times and present the median of the correspond metrics in Table B.3.

units	user [s]	kernel [s]	load	flows/s
2	29.1	1.1	198 %	93.5m
3	43.4	1.7	296 %	92.8m
4	56.0	4.2	393 %	65.9m
5	69.8	4.5	483 %	74.0m
6	78.0	8.1	558 %	80.1m
7	88.0	11.7	643 %	77.9m
8	94.5	18.1	724 %	69.5m

Table B.3: *Performance measurements for the ring topology*

We asked, first, if higher flow throughput can be achieved when file I/0 is reduced. In particular, is the throughput increased compared to 545k flows per second achieved by *reader_cc* on bzip2 compressed trace *midnight*? The answer is yes. As Table B.3 reveals, all ring topologies including 1 upto 16 units achieve at least a 100 times higher throughput.

Actually, the throughput performance deceases by building longer chains. This observation leads us to believe that different units have to fetch the memory across the CPUs using HyperTransport. Since the HyperTransprt Bus has only limited capacity, it becomes the new bottleneck resource of the system. But, very likely this number was bound by the selected traffic trace, file compression algorithm, and storage system. Nevertheless, this shows that FlowBox would allows us, in theory, to process the 330 Mio flows of the one hour trace *midday* within less than 6 seconds.

Bibliography

[1] K. Adamova. Security of virtual service migration in cloud infrastructures. Masterthesis, ETH Zurich, 2012.

[2] M. Allman, V. Paxson, and J. Terrell. A Brief History of Scanning. In *Proc. of ACM IMC*, 2007.

[3] Apple. iCloud security and privacy overview (HT4865). *support.apple.com*, Sept. 2012.

[4] Arbor Networks. Peakflow. `www.arbornetworks.com`.

[5] D. Aschwanden. Who turned off the Internet? Masterthesis, ETH Zurich, 2012.

[6] T. Auld, A. Moore, and S. Gull. Bayesian Neural Networks for Internet Traffic Classification. *IEEE Transactions on Neural Networks*, 2007.

[7] E. B. Claise. RFC5101 – Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information, 2008.

[8] T. Barseghian. Online Privacy: Kids Know More Than You Think. *PBS Mediashift*, 2012.

[9] G. Bartlett, J. Heidemann, and C. Papadopoulos. Understanding Passive and Active Service Discovery. In *Proc. of ACM IMC*, 2007.

[10] L. Bernaille, R. Teixeira, and K. Salamatian. Early application identification. In *Proc. of ACM CoNEXT*, 2006.

[11] R. Beverly and K. Sollins. Exploiting Transport-Level Characteristics of Spam. In *Proc. of CEAS*, 2008.

[12] Bitdefender. Trojan Now Uses Hotmail, Gmail as Spam Hosts. *www.bitdefender.com*, Aug. 2007.

[13] N. Bonelli, A. D. Pietro, S. Giordano, and G. Procissi. On Multi–gigabit Packet Capturing with Multi–core Commodity Hardware. In *Proc. of PAM*, 2012.

[14] C. Borgelt and R. Kruse. Induction of association rules: Apriori implementation. In *Proc. in Computational Statistics (Compstat)*, 2002.

[15] P. Borgnat, G. Dewaele, K. Fukuda, P. Abry, and K. Cho. Seven years and One Day: Sketching the Evolution of Internet Traffic. In *Proc. IEEE INFOCOM*, 2009.

[16] M. Brown. Pakistan hijacks YouTube. *www.renesys.com/blog*, 2008.

[17] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. *Network*, 2010.

[18] R. Bush, O. Maennel, M. Roughan, and S. Uhlig. Internet optometry: assessing the broken glasses in Internet reachability. In *Proc. of ACM IMC*, 2009.

[19] CA Technologies. NetQoS Performance Center.

[20] Campaign Monitor. Email client popularity. *www.campaignmonitor.com*, 2009.

[21] Carrier IQ. Mobile Intelligence. `www.carrieriq.com`.

[22] CERT NetSA Security Suite. Silk.

[23] C. Chang and C. Lin. Libsvm: a library for support vector machines.

[24] K. Cho. Broadband Traffic Report: Traffic Shifting away from P2P File Sharing to Web Services. Technical Report Internet Infrastructure Review vol.8, Internet Initiative Japan Inc., 2010.

[25] T. Choi, C. Kim, S. Yoon, J. Park, B. Lee, H. Kim, and H. Chung. Content-Aware Internet Application Traffic Measurement and Analysis. In *Proc. of IEEE/IFIP NOMS*, 2005.

[26] Cisco. Cisco Multi NetFlow Collector.

[27] Cisco. Network Management Case Study: How Cisco IT Uses NetFlow to Capture Network Behavior, Security, and Capacity Data, 2007.

[28] Cisco. Cisco visual networking index: Forecast and methodology, 2011-2016. Technical report, Cisco, 2012.

[29] Cisco. Introduction to Cisco IOS NetFlow - A Technical Overview. Technical report, Cisco, 2012.

[30] B. Claise, G. Sadasivan, V. Valluri, and M. Djernaes. RFC3954 – Cisco systems NetFlow services export version 9, 2004.

[31] R. Clayton. Using Early Results from the spamHINTS. In *Proc. of CEAS*, 2006.

[32] T. Cormen, C. Leiserson, R. Rivst, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[33] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli. Traffic Classification Through Simple Statistical Fingerprinting. *ACM SIGCOMM CCR*, 2007.

[34] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Trans. Networking*, 1997.

[35] L. Deri and S. Suin. Effective traffic measurement using ntop. *IEEE Communication Magazine*, 2000.

[36] P. Desikan and J. Srivastava. Analyzing Network Traffic to Detect E-Mail Spamming Machines. In *ICDM Workshop on Privacy and Security Aspects of Data Mining*, 2004.

[37] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In *Proc. of USENIX*, 2006.

[38] Dropbox Inc. Help center: How secure is dropbox? *www.dropbox.com*, Oct. 2012.

[39] Z. Duan, K. Gopalan, and X. Yuan. Behavioral Characteristics of Spammers and Their Network Reachability Properties. In *Proc. of IEEE ICC*, 2007.

[40] Electronic Frontier Foundation. HTTPS Everywhere. `www.eff.org/https-everywhere`.

[41] Electronic Frontier Foundation. Privacy. *www.eff.org/issues/privacy*, 2012.

[42] C. Estan, S. Savage, and G. Varghese. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *Proc. of ACM SIGCOMM*, 2003.

[43] Facebook. `www.facebook.com`.

[44] T. Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 2006.

[45] N. Feamster, D. Anderson, H. Balakrishnan, and F. Kaashoek. Measuring the Effects of Internet Path Faults on Reactive Routing. In *Proc. of ACM SIGMETRICS*, 2003.

[46] A. Feldmann, O. Maennel, M. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *Proc. of ACM SIGCOMM*, 2004.

[47] P. Gamble and P. Gamble. *Knowledge Management: A State-Of-The-Art Guide*. Kogan Page, 2002.

[48] German Federal Office for Information Security. Critical infrastructures, 2011.

[49] E. Glatzand and X. Dimitropoulos. Classifying internet one-way traffic. In *Proc. of ACM IMC*, 2012.

[50] L. Gomes, R. Almeida, L. Bettencourt, V. Almeida, and J. Almeida. Comparative Graph Theoretical Characterization of Networks of Spam and Legitimate Email. *Arxiv physics/0504025*, 2005.

[51] L. Gomes, C. Cazita, J. Almeida, V. Almeida, and W. Meira. Characterizing a spam traffic. In *Proc. of ACM IMC*, 2004.

[52] Google. `www.google.com`.

[53] Google. Google Drive. `www.drive.google.com`.

[54] Google. YouTube. `www.youtube.com`.

[55] S. Gorman and J. Barnes. Cyber Combat: Act of War. *The Wallstreet Journal*, 2011.

[56] Gu, G., Perdisci, R., Zhang, J., Lee, W. BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet Detection. In *Proc. of USENIX Security*, 2008.

[57] P. Haag. Watch your Flows with NfSen and NFDUMP. In *Proc. of RIPE Meeting*, 2005.

[58] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. Acas: Automated Construction of Application Signatures. In *Proc. of SIGCOMM MineNet Workshop*, 2005.

[59] S. Hao, N. Feamster, A. Gray, N. Syed, and S. Krasser. Detecting Spammers with SNARE: Spatio-Temporal Network-level Automatic Reputation Engine. In *Proc. of USENIX*, 2009.

[60] E. Harris. The next step in the spam control war: Greylisting. *projects.puremagic.com/greylisting*, 2003.

[61] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, and G. Varghese. Network Monitoring Using Traffic Dispersion Graphs. In *Proc. of ACM IMC*, 2007.

[62] Internet World Statistics. Users and Population. Technical Report 1-December, Miniwatts Marketing Group, 2011.

[63] IRONPORT. Internet security trends. Technical report, Cisco, 2008.

[64] IsarFlow. IsarFlow Network Monitoring Tool. `isarflow.com`.

[65] J. Hoover. Security measures to protect Ubuntu One subscriber. *wiki.ubuntu.com*, Mar. 2011.

[66] J.Wu, M. Mao, J. Rexford, and J. Wang. Finding a Needle in a Haystack: Pinpointing Significant BGP Routing Changes in an IP Network. In *Proc. NSDI*, 2005.

[67] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. In *Proc. of ACM SIGCOMM*, 2005.

[68] T. Karagiannis, K. Papagiannaki, N. Taft, and M. Faloutsos. Profiling the End Host. In *Proc. of PAM*, 2007.

[69] E. Katz-Bassett, H. Madhyastha, J. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying Black Holes in the Internet with Hubble. In *Proc. NSDI*, 2008.

[70] G. Keizer. Scammers Exploit Public Lists of Hijacked Hotmail Passwords. *www.computerworld.com*, Oct. 2009.

[71] H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee. Internet Traffic Classification Demystified: Myths, Caveats, and the Best Practices. In *Proc. of ACM CoNEXT*, 2008.

[72] H. Kim, M. Fomenkov, K. Claffy, N. Brownlee, D. Barman, and M. Faloutsos. Comparison of Internet Traffic Classification Tools. In *Proc. of IMRG workshop on application classification and identification report*, 2009.

[73] S. Kipp. Exponential bandwidth growth and cost declines. *www.networkworld.com*, Apr. 2012.

[74] T. Kleefass. Passively detecting remote connectivity issues using flow accounting. Diplomarbeit, University Stuttgart, 2009.

[75] J. Klensin. Rfc 2821 – simple mail transfer protocol, 2001.

[76] J. Kögel and S. Scholz. Processing of flow accounting data in Java: framework design and performance evaluation. In *Proc. of EUNICE*, 2010.

[77] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the edge network. In *Proc. of ACM IMC*, 2010.

[78] S. Kremp. Pentagon schliesst Cyber-Angriffe durch die USA nicht aus. *www.heise.de*, 2011.

[79] S. Kremp. Facebook muss neue Nutzungsbedingungen überprüfen. *www.heise.de*, 2012.

[80] J. Kristoff. Botnets. In *NANOG32*, 2004.

[81] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. Internet Inter-Domain Traffic. In *Proc. of ACM SIGCOMM*, 2010.

[82] Lancope. StealthWatch. `www.lancope.com`.

[83] F. Lawler. The 10 Most Bizarre and Annoying Causes of Fiber Cuts. *blog.level3.com*, 2011.

[84] S. Leinen. Fluxoscope: a System for Flow-based Accounting. Technical Report CATI-SWI-IM-P-000-0.4, CATI, 2000.

[85] D. Leonard and D. Loguinov. Demystifying Service Discovery: Implementing and Internet-Wide Scanner. In *Proc. of ACM IMC*, 2010.

[86] W. Li, A. W. Moore, and M. Canini. Classifying HTTP Traffic in the New Age. In *ACM SIGCOMM, Poster Session*, 2008.

[87] Z. Li, R. Yuan, and X. Guan. Accurate Classification of the Internet Traffic Based on the SVM Method. In *Proc. of IEEE ICC*, 2007.

[88] H. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: an information plane for distributed services. In *Proc. of Symposium on Operating Systems Design and Implementation*, 2006.

[89] D. Madory. Hurricane Sandy: Initial Impact. *www.renesys.com/blog*, 2012.

[90] A. Mahimkar, Z. Ge, J. Wang, J. Yates, Y. Zhang, and M. S. J. Emmons, B. Huntley. Rapid detection of maintenance induced changes in service performance. In *Proc. of ACM CoNEXT*, 2011.

[91] A. Mahimkar, H. Song, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and J. Emmons. Detecting the performance impact of upgrades in large operational networks. *ACM SIGCOMM CCR*, 2010.

[92] G. Maier, A. Feldmann, V. Paxson, and M. Allman. On Dominant Characteristics of Residential Broadband Internet Traffic. In *Proc. of ACM IMC*, 2009.

[93] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[94] MaxMind. IP Intelligence Solution. www.maxmind.com.

[95] A. McGregor, M. Hall, P. Lorier, and J. Brunskill. Flow Clustering Using Maching Learning Techniques. In *Proc. of PAM*, 2004.

[96] Microsoft. Microsoft Outlook Web Access. www.microsoft.com/exchange/en-us/outlook-web-app.aspx.

[97] A. Mitseva. On-line monitoring of cloud providers with nfsen. Bachelorthesis, Technical University of Sofia, 2012.

[98] A. Moore and D. Zuev. Internet Traffic Classification Using Bayesian Analysis Techniques. In *Proc. of ACM SIGMETRICS*, 2005.

[99] A. W. Moore and P. Konstantina. Toward the Accurate Identification of Network Applications. In *Proc. of PAM*, 2005.

[100] J. Myers and A. Well. *Research Design and Statistical Analysis*. Routledge, 2002.

[101] Nagios Enterprises. www.nagios.org.

[102] North American Network Operators' Group. www.nanog.org.

[103] R. Olsen. *Design Patterns in Ruby*. Addison-Wesley Professional, 2007.

[104] OpenFlow. Enabling Innovation in Your Network. http://www.openflow.org.

[105] outages@outages.org. The outages mailinglist. http://puck.nether.net/mailman/listinfo/outages.

[106] V. Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM Trans. Networking*, 1997.

[107] M. Pietrzyk, J.-L. Costeux, G. Urvoy-Keller, and T. En-Najjary. Challenging Statistical Classification for Operational Usage: the ADSL Case. In *Proc. of ACM IMC*, 2009.

[108] M. Pietrzyk, T. En-Najjary, G. Urvoy-Keller, and J. Costeux. Hybrid Traffic Identification. Technical report, Institut Eurecom, 2010.

[109] L. Quan and J. Heidemanm. On the Characteristics and Reasons of Long-lived Internet Flows. In *Proc. of ACM IMC*, 2010.

[110] A. Ramachandran, D. Dagon, and N. Feamster. Can DNS-based blacklists keep up with bots. In *Proc. of CEAS*, 2006.

[111] A. Ramachandran and N. Feamster. Understanding the Network-Level Behavior of Spammers. In *Proc. of ACM SIGCOMM*, 2006.

[112] A. Ramachandran, N. Feamster, and S. Vempala. Filtering Spam with Behavioral Blacklisting. In *Proc. ACM CCS*, 2007.

[113] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani. Fast Monitoring of Traffic Subpopulations. In *Proc. of ACM IMC*, 2008.

[114] J. Rao and D. Reiley. The Economics of Spam. *The Journal of Economic Perspectives*, 2012.

[115] A. Rice. Facebook: A Continued Commitment to Security. *The Facebook Blog*, Jan. 2011.

[116] J. Riden. How fast-flux service networks work. *www.honeynet.org*, Aug. 2008.

[117] E. Romijn. RIPE/Duke event. *labs.ripe.net*, Aug. 2010.

[118] M. Roughan, S. Sen, O. Spatschek, and N. Duffield. Class-of-Service Mapping for QoS: a Statistical Signature-Based Approach to IP Traffic Classification. In *Proc. of ACM IMC*, 2004.

[119] J. Rowley. The wisdom hierarchy: representations of the DIKW hierarchy. *Journal of Information Science*, 2007.

[120] J. Rowley. *Organizing knowledge: an introduction to managing access to information*. Ashgate Publishing Company, 2008.

[121] G. Sadasivan, N. Brownlee, B. Claise, and J. Quittek. RFC5470 – Architecture for IP flow information export, 2006.

[122] K. Scarfone and P. Mell. Guide to Intrusion Detection and Prevention Systems (IDPS). Technical Report SP 800-94, NIST, 2007.

[123] D. Schatzmann, M. Burkhart, and T. Spyropoulos. Flow-level Characteristics of Spam and Ham. Technical Report TIK-Report-291, ETH Zurich, 2008.

[124] D. Schatzmann, M. Burkhart, and T. Spyropoulos. Inferring Spammers in the Network Core. In *Proc. of PAM*, 2009.

[125] D. Schatzmann, S. Leinen, J. Kögel, and W. Mühlbauer. FACT: Flow-Based Approach for Connectivity Tracking. In *Proc. of PAM*, 2011.

[126] D. Schatzmann and W. Mühlbauer. Tracing Network Services. Technical Report TIK-Report-338, ETH Zurich, 2011.

[127] D. Schatzmann, W. Mühlbauer, T. Spyropoulos, and X. Dimitropoulos. Digging into HTTPS: Flow-Based Classification of Webmail Traffic . In *Proc. of ACM IMC*, 2010.

[128] D. Schatzmann, W. Mühlbauer, B. Trammell, and K. Salamatian. Scaling traffic classification through spatio-temporal sampling. In *Under Submission*, 2013.

[129] S. Schillace. Default HTTPS access for Gmail. *gmail-blog.blogspot.com*, Jan. 2010.

[130] F. Schneider, S. Agarwal, T. Alpcan, and A. Feldmann. The New Web: Characterizing AJAX Traffic. In *Proc. of PAM*, 2008.

[131] M. Schwartz. Obama's Consumer Privacy Bill of Rights: 9 Facts. *www.informationweek.com*, Feb. 2012.

[132] S. Sen, O. Spatscheck, and D. Wang. Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures. In *Proc. of WWW*, 2004.

[133] SpamCop. The SpamCop Blocking List. `www.spamcop.net`.

[134] Spamhaus. The Policy Block List. `www.spamhaus.org/pbl`.

[135] Spamhaus. The Spamhaus Block List. `www.spamhaus.org/sbl`.

[136] Swiss Federal Office for Civil Protection). Critical infrastructure protection, 2011.

[137] N. Syed, N. Feamster, A. Gray, and S. Krasser. SNARE: Spatio-temporal Network-level Automatic Reputation Engine. Technical Report GT-CSE-08-02, Georgia Tech, 2008.

[138] M. Taylor. Broadband is power. *blog.level3.com*, 2011.

[139] Team Cymru Community Services. The Bogon Reference. `www.team-cymru.org/Services/Bogons`.

[140] B. Tellenbach, M. Burkhart, D. Schatzmann, D. Gugelmann, and D. Sornette. Accurate network anomaly classification with generalized entropy metrics. *Computer Networks*, 2011.

[141] B. Tellenbach, M. Burkhart, D. Sornette, and T. Maillart. Beyond Shannon: Characterizing Internet Traffic with Generalized Entropy Metrics. In *Proc. of PAM*, 2009.

[142] The Horde Project. Horde. `www.horde.org`.

[143] The Swiss Education and Research Network. `www.switch.ch`.

[144] The World Bank Group. Indicators – Internet users. Technical Report 1-December, The World Bank Open Data, 2011.

[145] K. Thompson, G. Miller, and R. Wilder. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network*, 1997.

[146] B. Trammell, E. Boschi, G. Procissi, C. Callegari, P. Dorfinger, and D. Schatzmann. Identifying Skype traffic in a large-scale flow data repository. In *Proc. of TMA Workshop*, 2011.

[147] B. Trammell, B. Tellenbach, D. Schatzmann, and M. Burkhart. Peeling Away Timing Error in NetFlow Data. In *Proc. of PAM*, 2011.

[148] Twitter. HelpCenter: Safety: Keeping Your Account Secure. *support.twitter.com*, Jan. 2011.

[149] S. Webster, R. Lippmann, and M. Zissman. Experience Using Active and Passive Mapping for Network Situational Awareness. In *Proc. IEEE NCA*, 2006.

[150] Wikipedia. Arpanet — wikipedia, the free encyclopedia, 2011.

[151] Wikipedia. Internet — wikipedia, the free encyclopedia, 2011.

[152] Wikipedia. Dikw pyramid — wikipedia, the free encyclopedia, 2012.

[153] Wikipedia. List of tcp and udp port numbers — wikipedia, the free encyclopedia, 2012.

[154] Wikipedia. Pipeline (software) — wikipedia, the free encyclopedia, 2012.

[155] Wikipedia. Spam (electronic) — wikipedia, the free encyclopedia, 2012.

[156] A. Wilkens. Bericht: USA wollen Hackerangriffe zum Kriegsgrund erklären. *www.heise.de*, 2011.

[157] Y. Won, B. Park, H. Ju, M. Kim, and J. Hong. A Hybrid Approach for Accurate Application Traffic Identification. In *Proc. of IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, 2006.

[158] Wong, M., Schlitt, W. RFC 4408 – Sender Policy Framework (SPF), 2006.

[159] E. Wustrow, M. Karir, M. Bailey, F. Jahanian, and G. Huston. Internet background radiation revisited. In *Proc. of ACM IMC*, 2010.

[160] Yahoo. The Middleware Threats. *biz.yahoo.com*, 1999.

[161] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *Proc. of Symposium on Operating Systems Design and Implementation*, 2004.

[162] Y. Zhang, M. Mao, and M. Zhang. Effective Diagnosis of Routing Disruptions from End Systems. In *Proc. NSDI*, 2008.

Acknowledgments

I am deeply grateful to Prof. Dr. Bernhard Plattner for his invaluable support during this thesis and for providing the freedom to develop and explore my own visions and ideas. Special thanks to Wolfgang Mühlbauer and Thrasyvoulos Spyropoulos for supervising this work and for their great commitment and brilliant support.

I like to extend my thanks to Simon Leinen for providing very interesting insights into the daily business of an ISP and Kavé Salamatian for opening new horizons in the field of information theory. Moreover, I thank Prof. Dr. Rolf Stadler and Xenofontas Dimitropoulos for being co-examiners of this thesis and for providing very valuable feedback.

A special thanks to my office mates Martin Burkhart and David Gugelmann for their support through all the ups and downs of PhD student life.

I am grateful to my collaborators and coauthors Bernhard Ager, Bernhard Distl, Stefan Frei, Theus Hossmann, Jochen Kögel, Franck Legendre, Vincent Lenders, Stephan Neuhaus, Bernhard Tellenbach, Brian Trammell, and Sacha Trifunovic.

Further, I would like to thank all my colleagues from the Communication Systems Group (CSG) who have not been mentioned so far, in particular Abdullah Alhussainy, Panayotis Antoniadis, Mahdi Asadpour, Rainer Baumann, Ehud Ben Porat, Daniel Borkmann, Daniela Brauckhoff, Paolo Carta, Thomas Dübendorfer, Francesco Fusco, Domenico Giustiniano, Eduard Glatz, Simon Heimlicher, Karin Anna Hummel, Merkourios Karaliopoulos, Ariane Keller, Vasileios Kotronis, Maciej Kurant, Jose Mingorance-Puga, Andreea Picu, Gabriel Popa, Ilias Raftopoulos, Sacha Trifunovic, Arno Wagner, Deng Wenping, and Jinyao Yan.

Furthermore, I thank Caterina Sposato, Tanja Lantz, Beat Futterknecht, Thomas Steingruber, and Damian Friedli for providing efficient administra-

tive and technical support.

In addition, I want to thank the students I supervised for their contributions to my research, namely Daniel Aschwanden, Aarno Aukia, Rene Buehlmann, Adrian Gämperli, Guido Hungerbühler, Pablo Schläpfer, Mathias Schnydrig, and Manuel Widmer.

Last but not least, I am deeply grateful to my parents, Ruth and Herbert, my daughter Julia, and my wife Martina for their encouragement and whose patient love enabled me to complete this work.